Machine Learing Compiler

Release 0.0.1

Tianqi Chen, Siyuan Feng, Ruihang Lai, Hongyi Jin

Jul 10, 2025

Contents

1 Introduction **3**
- 1.1 What is ML Compilation . . . 4
- 1.2 Why Study ML Compilation . . . 5
- 1.3 Key Elements of ML Compilation . . . 6
- 1.4 Summary . . . 8

2 Tensor Program Abstraction **9**
- 2.1 Primitive Tensor Function . . . 9
- 2.2 Tensor Program Abstraction . . . 10
- 2.3 Summary . . . 12
- 2.4 TensorIR: Tensor Program Abstraction Case Study . . . 12
- 2.5 Exercises for TensorIR . . . 28

3 End to End Model Execution **35**
- 3.1 Prelude . . . 35
- 3.2 Preparations . . . 35
- 3.3 End to End Model Integration . . . 38
- 3.4 Constructing an End to End IRModule in TVMScript . . . 40
- 3.5 Build and Run the Model . . . 45
- 3.6 Integrate Existing Libraries in the Environment . . . 46
- 3.7 Mixing TensorIR Code and Libraries . . . 48
- 3.8 Bind Parameters to IRModule . . . 49
- 3.9 Discussions . . . 49
- 3.10 Summary . . . 50

4 Automatic Program Optimization **51**
- 4.1 Prelude . . . 51
- 4.2 Preparations . . . 51
- 4.3 Recap: Transform a Primitive Tensor Function. . . . 52
- 4.4 Stochastic Schedule Transformation . . . 53
- 4.5 Search Over Stochastic Transformations . . . 57
- 4.6 Putting Things Back to End to End Model Execution . . . 66
- 4.7 Discussions . . . 72
- 4.8 Summary . . . 73

5 Integration with Machine Learning Frameworks **75**
- 5.1 Prelude . . . 75
- 5.2 Preparations . . . 75
- 5.3 Build an IRModule Through a Builder . . . 75
- 5.4 Import Model From PyTorch . . . 79
- 5.5 Coming back to FashionMNIST Example . . . 82

5.6 Remark: Translating into High-level Operators . . . 86
5.7 Discussions . . . 87
5.8 Summary . . . 87

6 GPU and Hardware Acceleration **89**
6.1 Part 1 . . . 89
6.2 Part 2 . . . 97

7 Computational Graph Optimization **107**
7.1 Prelude . . . 107
7.2 Preparations . . . 107
7.3 Pattern Match and Rewriting . . . 108
7.4 Fuse Linear and ReLU . . . 111
7.5 Map to TensorIR Calls . . . 113
7.6 Build and Run . . . 115
7.7 Discussion . . . 116
7.8 Summary . . . 117

Deploying innovative AI models in different production environments becomes a common problem as AI applications become more ubiquitous in our daily lives. Deployment of both training and inference workloads bring great challenges as we start to support a combinatorial choice of models and environment. Additionally, real world applications bring with a multitude of goals, such as minimizing dependencies, broader model coverage, leveraging the emerging hardware primitives for performance, reducing memory footprint, and scaling to larger environments.

Solving these problems for training and inference involves a combination of ML programming abstractions, learning-driven search, compilation, and optimized library runtime. These themes form an emerging topic – machine learning compilation that contains active ongoing developments. In this tutorials sequence, we offer the first comprehensive treatment of its kind to study key elements in this emerging field systematically. We will learn the key abstractions to represent machine learning programs, automatic optimization techniques, and approaches to optimize dependency, memory, and performance in end-to-end machine learning deployment.

This material serves as the reference for MLC course, we will populate notes and tutorials here as course progresses.

1 | Introduction

Machine learning applications have undoubtedly become ubiquitous. We get smart home devices powered by natural language processing and speech recognition models, computer vision models serve as backbones in autonomous driving, and recommender systems help us discover new content as we explore. Observing the rich environments where AI apps run is also quite fun. Recommender systems are usually deployed on the cloud platforms by the companies that provide the services. When we talk about autonomous driving, the natural things that pop up in our heads are powerful GPUs or specialized computing devices on vehicles. We use intelligent applications on our phones to recognize flowers in our garden and how to tend them. An increasing amount of IoT sensors also come with AI built into those tiny chips. If we drill down deeper into those environments, there are an even greater amount of diversities involved. Even for environments that belong to the same category(e.g. cloud), there are questions about the hardware(ARM or x86), operation system, container execution environment, runtime library variants, or the kind of accelerators involved. Quite some heavy liftings are needed to bring a smart machine learning model from the development phase to these production environments. Even for the environments that we are most familiar with (e.g. on GPUs), extending machine learning models to use a non-standard set of operations would involve a good amount of engineering. Many of the above examples are related to machine learning inference — the process of making predictions after obtaining model weights. We also start to see an important trend of deploying training processes themselves onto different environments. These applications come from the need to keep model updates local to users' devices for privacy protection reasons or scaling the learning of models onto a distributed cluster of nodes. The different modeling choices and inference/training scenarios add even more complexity to the productionisation of machine learning.

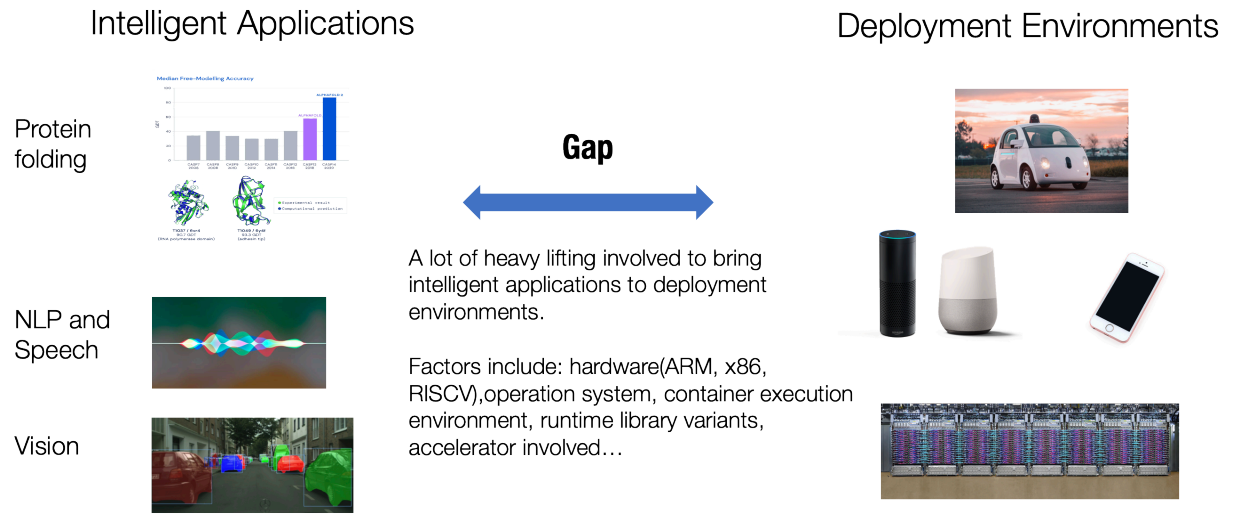


Fig. 1.1: Gap in ML deployment.

This course studies the topic of bringing machine learning from the development phase to production envi-

ronments. We will study a collection of methods that facilitate the process of ML productionisation. Machine learning productionisation is still an open and active field, with new techniques being developed by the machine learning and systems community. Nevertheless, we start to see common themes appearing, which end up in the theme of this course.

1.1 What is ML Compilation

Machine learning compilation (MLC) is the process of transforming and optimizing machine learning execution from its development form to its deployment form.

Development form refers to the set of elements we use when developing machine learning models. A typical development form involves model descriptions written in common frameworks such as PyTorch, TensorFlow, or JAX, as well as weights associated with them.

Deployment form refers to the set of elements needed to execute the machine learning applications. It typically involves a set of code generated to support each step of the machine learning model, routines to manage resources (e.g. memory), and interfaces to application development environments (e.g. java API for android apps).

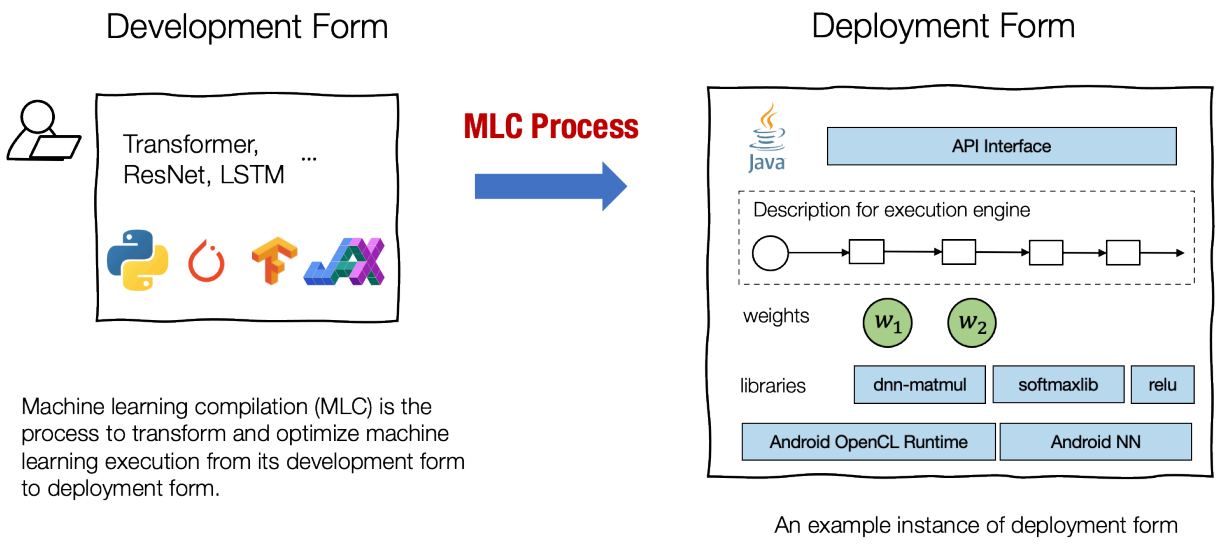


Fig. 1.1.1: Development and Deployment Forms.

We use the term "compilation" as the process can be viewed in close analogy to what traditional compilers do — a compiler takes our applications in development form and compiles them to libraries that can be deployed. However, machine learning compilation still differs from traditional compilation in many ways.

First of all, this process does not necessarily involve code generation. For example, the deployment form can be a set of pre-defined library functions, and the ML compilation only translates the development forms onto calls into those libraries. The set of challenges and solutions are also quite different. That is why studying machine learning compilation as its own topic is worthwhile, independent of a traditional compilation. Nevertheless, we will also find some useful traditional compilation concepts in machine learning compilation.

The machine learning compilation process usually comes with the several goals:

Integration and dependency minimization. The process of deployment usually involves integration — assembling necessary elements together for the deployment app. For example, if we want to enable an android

camera app to classify flowers, we will need to assemble the necessary code that runs the flower classification models, but not necessarily other parts that are not related to the model (e.g. we do not need to include an embedding table lookup code for NLP applications). The ability to assemble and minimize the necessary dependencies is quite important to reduce the overall size and increase the possible number of environments that the app can be deployed to.

Leveraging hardware native acceleration. Each deployment environment comes with its own set of native acceleration techniques, many of which are especially developed for ML. One goal of the machine learning compilation process is to leverage that hardware's native acceleration. We can do it through building deployment forms that invoke native acceleration libraries or generate code that leverages native instructions such as TensorCore.

Optimization in general. There are many equivalent ways to run the same model execution. The common theme of MLC is optimization in different forms to transform the model execution in ways that minimize memory usage or improve execution efficiency.

There is not a strict boundary in those goals. For example, integration and hardware acceleration can also be viewed as optimization in general. Depending on the specific application scenario, we might be interested in some pairs of source models and production environments, or we could be interested in deploying to multiple and picking the most cost-effective variants.

Importantly, MLC does not necessarily indicate a single stable solution. As a matter of fact, many MLC practices involves collaborations with developers from different background as the amount of hardware and model set grows. Hardware developers need support for their latest hardware native acceleration, machine learning engineers aim to enable additional optimizations, and scientists bring in new models.

1.2 Why Study ML Compilation

This course teaches machine learning compilation as a methodology and collections of tools that come along with the common methodology. These tools can work with or simply work inside common machine learning systems to provide value to the users. For machine learning engineers who are working on ML in the wild, MLC provides the bread and butter to solve problems in a principled fashion. It helps to answer questions like what methodology we can take to improve the deployment and memory efficiency of a particular model of interest and how to generalize the experience of optimizing a single part of the model to a more generic end-to-end solution. For machine learning scientists, MLC offers a more in-depth view of the steps needed to bring models into production. Some of the complexity is hidden by machine learning frameworks themselves, but challenges remain as we start to incorporate novel model customization or when we push our models to platforms that are not well supported by the frameworks. ML compilation also gives ML scientists an opportunity to understand the rationales under the hood and answer questions like why my model isn't running as fast as expected and what can be done to make the deployment more effective. For hardware providers, MLC provides a general approach to building a machine learning software stack to best leverage the hardware they build. It also provides tools to automate the software optimizations to keep up with new generations of hardware and model developments while minimizing the overall engineering effort. Importantly, machine learning compilation techniques are not being used in isolation. Many of the MLC techniques have been applied or are being incorporated into common machine learning frameworks, and machine learning deployment flows. MLC is playing an increasingly important role in shaping the API, architectures, and connection components of the machine learning software ecosystem. Finally, learning MLC itself is fun. With the set of modern machine learning compilation tools, we can get into stages of machine learning model from high-level, code optimizations, to bare metal. It is really fun to get end to end understanding of what is happening here and use them to solve our problems.

1.3 Key Elements of ML Compilation

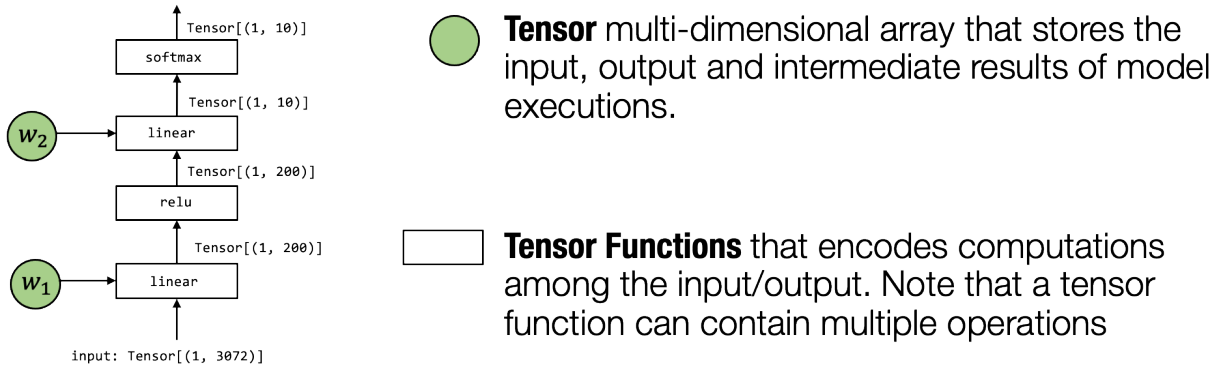


Fig. 1.3.1: MLC Elements.

In the previous sections, we discussed machine learning compilation at a high level. Now, let us dive deeper into some of the key elements of machine learning compilation. Let us begin by reviewing an example of two-layer neural network model execution.

In this particular model, we take a vector by flattening pixels in an input image; then, we apply a linear transformation that projects the input image onto a vector of length 200 with `relu` activation. Finally, we map it to a vector of length 10, with each element of the vector corresponding to how likely the image belongs to that particular class.

Tensor is the first and foremost important element in the execution. A tensor is a multidimensional array representing the input, output, and intermediate results of a neural network model execution.

Tensor functions The neural network's "knowledge" is encoded in the weights and the sequence of computations that takes in tensors and output tensors. We call these computations tensor functions. Notably, a tensor function does not need to correspond to a single step of neural network computation. Part of the computation or entire end-to-end computation can also be seen as a tensor function.

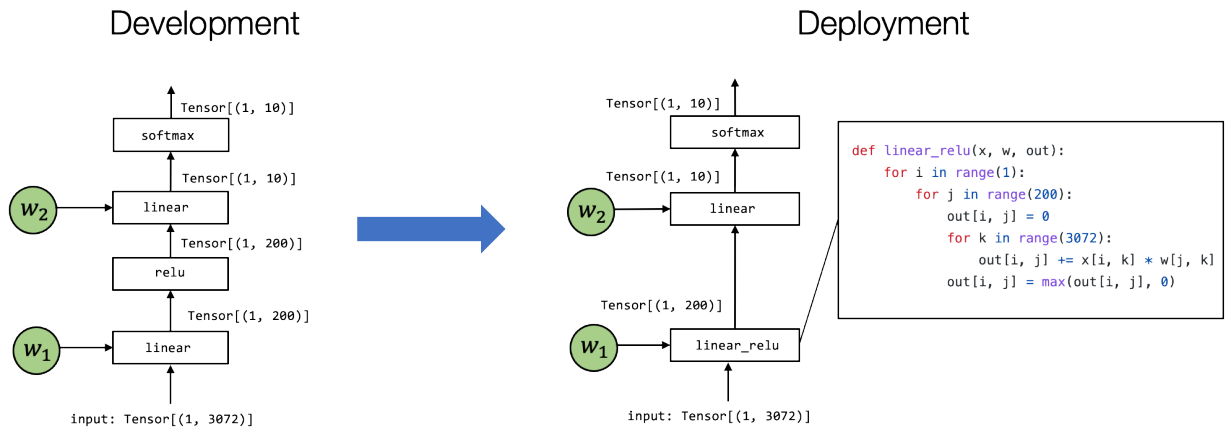


Fig. 1.3.2: Example MLC Process as Tensor Function Transformations.

There are multiple ways to implement the model execution in a particular environment of interest. The above

examples show one example. Notably, there are two differences: First, the first linear and relu computation are folded into a `linear_relu` function. There is now a detailed implementation of the particular linear_relu. Of course, the real-world use cases, the `linear_relu` will be implemented using all kinds of code optimization techniques, some of which will be covered in the later part of the lecture. MLC is a process of transforming something on the left to the right-hand side. In different settings, this might be done by hand, with some automatic translation tools, or both.

1.3.1 Remark: Abstraction and Implementations

One thing that we might notice is that we use several different ways to represent a tensor function. For example, `linear_relu` is shown that it can be represented as a compact box in a graph or a loop nest representation.

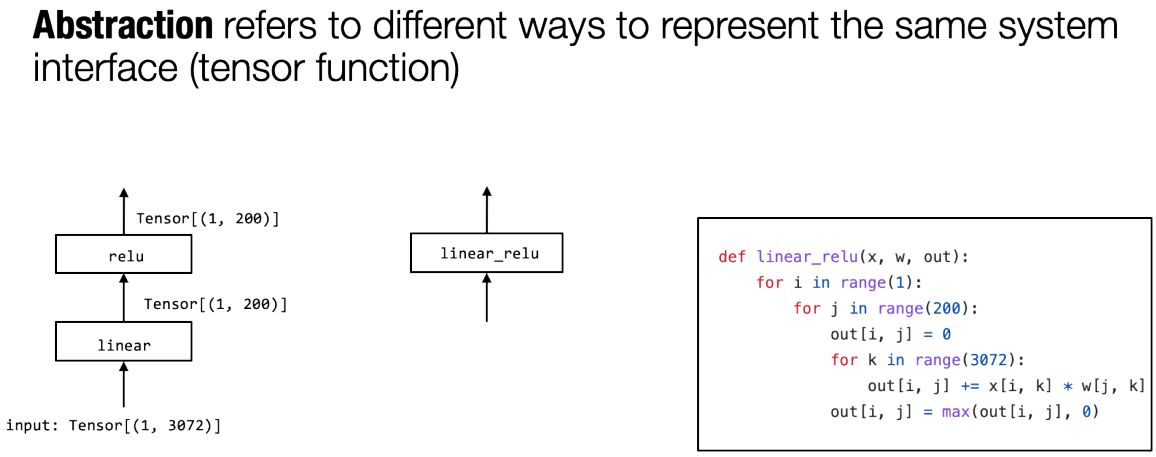


Fig. 1.3.3: Abstractions and Implementations.

We use **abstractions** to denote the ways we use to represent the same tensor function. Different abstractions may specify some details while leaving out other **implementation** details. For example, `linear_relu` can be implemented using another different for loops.

Abstraction and **implementation** are perhaps the most important keywords in all computer systems. An abstraction specifies "what" to do, and implementation provides "how" to do it. There are no specific boundaries. Depending on how we see it, the for loop itself can be viewed as an abstraction since it can be implemented using a python interpreter or compiled to a native assembly code.

MLC is effectively a process of transforming and assembling tensor functions under the same or different abstractions. We will study different kinds of abstractions for tensor functions and how they can work together to solve the challenges in machine learning deployment.

1.4 Summary

- Goals of machine learning compilation
  - Integration and dependency minimization
  - Leveraging hardware native acceleration
  - Optimization in general
- Why study ML compilation
  - Build ML deployment solutions.
  - In-depth view of existing ML frameworks.
  - Build up software stack for emerging hardware.
- Key elements of ML compilation
  - Tensor and tensor functions.
  - Abstraction and implementation are useful tools to think

2 | Tensor Program Abstraction

In this chapter, we will discuss the abstractions for a single "unit" step of computation and possible MLC transformations in these abstractions.

2.1 Primitive Tensor Function

The introductory overview showed that the MLC process could be viewed as transformations among tensor functions. A typical model execution involves several computation steps that transform tensors from input to the final prediction, and each unit step is called a primitive tensor function.

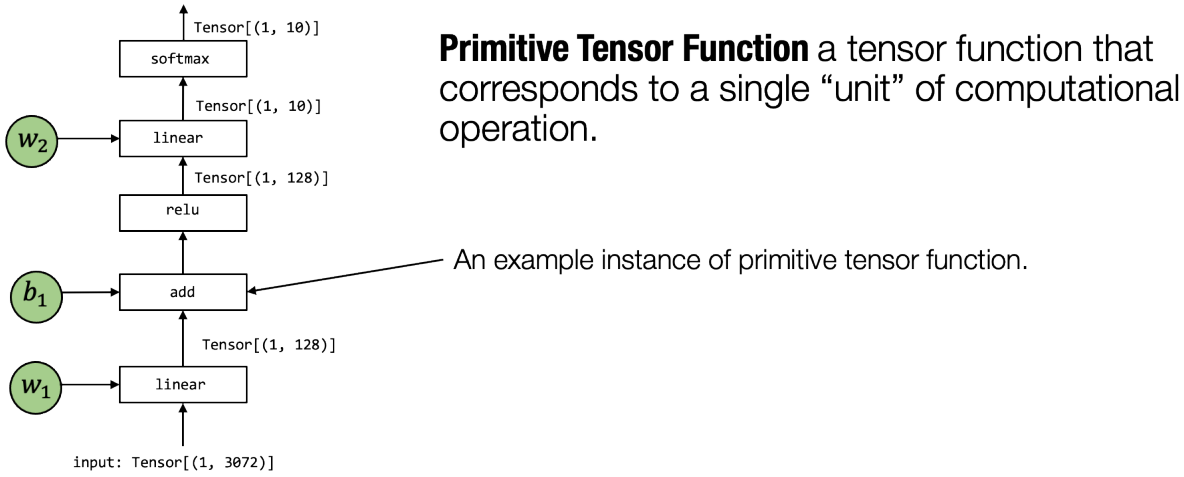


Fig. 2.1.1: Primitive Tensor Function

In the above figure, the tensor operator linear, add, relu, and softmax are all primitive tensor functions. Notably, many different abstractions can represent (and implement) the same primitive tensor function add (as shown in the figure below). We can choose to call into pre-built framework libraries(e.g. torch.add or numpy.add), and leverage an implementation in python. In practice, primitive functions are implemented in low-level languages such as C/C++ with sometimes a mixture of assembly code.

```
torch.add
```

```
def add(a, b, c):
    for i in range(128):
        c[i] = a[i] + b[i]
```

```
void add(float* a, float* b, float* c) {
  for(int i = 0; i < 128; ++i) {
    c[i] = a[i] + b[i];
  }
}
```

Fig. 2.1.2: Different forms of the same primitive tensor function

Many frameworks offer machine learning compilation procedures to transform primitive tensor functions into more specialized ones for the particular workload and deployment environment.

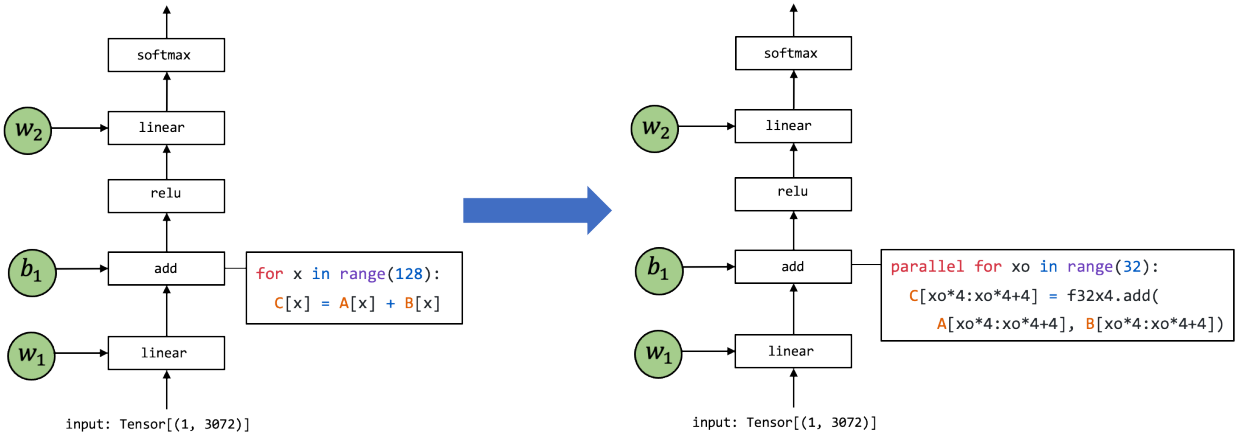


Fig. 2.1.3: Transformations between primitive tensor functions

The above figure shows an example where the implementation of the primitive tensor function add gets transformed into a different implementation. The particular code on the right is a pseudo-code representing possible set optimizations: the loop gets split into units of length `4` where `f32x4` add corresponds to a special vector add function that carries out the computation.

2.2 Tensor Program Abstraction

The last section talks about the need to transform primitive tensor functions. In order for us to effectively do so, we need an effective abstraction to represent the programs.

Usually, a typical abstraction for primitive tensor function implementation contains the following elements: multi-dimensional buffers, loop nests that drive the tensor computations, and finally, the compute statements themselves.

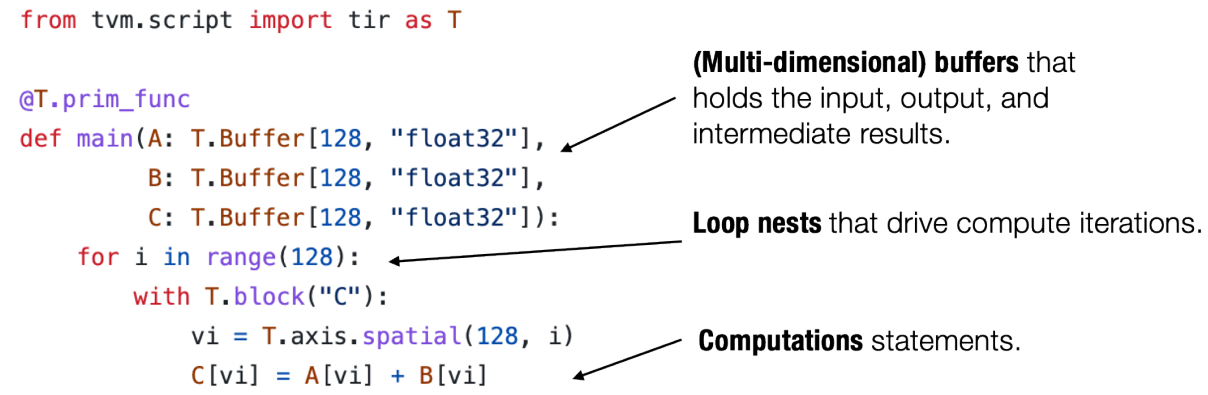


Fig. 2.2.1: The typical elements in a primitive tensor function

We call this type of abstraction tensor program abstraction. One important property of tensor program abstraction is the ability to change the program through a sequence of transformations pragmatically.

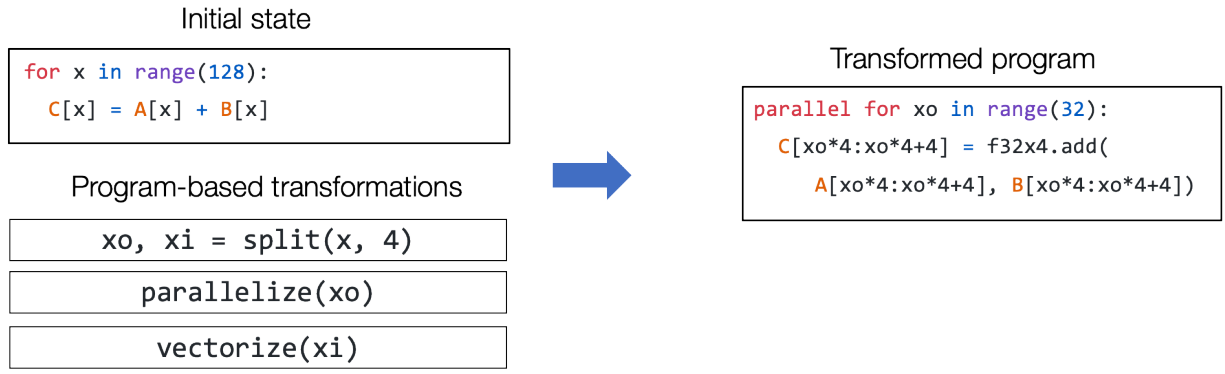


Fig. 2.2.2: Sequential transformations on a primitive tensor function

For example, we should be able to use a set of transformation primitives(split, parallelize, vectorize) to take the initial loop program and transform it into the program on the right-hand side.

2.2.1 Extra Structure in Tensor Program Abstraction

Importantly, we cannot perform arbitrary transformations on the program as some computations depend on the order of the loop. Luckily, most primitive tensor functions we are interested in have good properties (such as independence among loop iterations).

Tensor programs can incorporate this extra information as part of the program to facilitate program transformations.

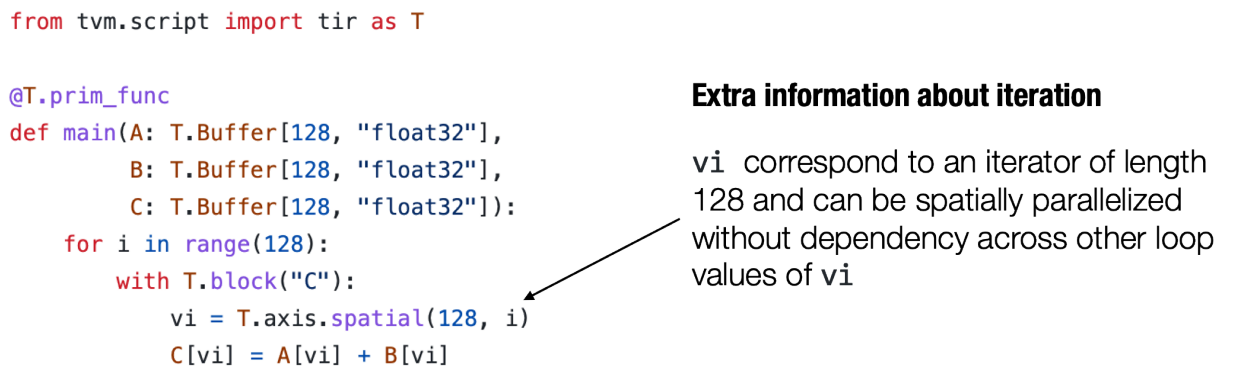


Fig. 2.2.3: Iteration is the extra information for tensor programs

For example, the above program contains the additional `T.axis.spatial` annotation, which shows that the particular variable `vi` is mapped to `i`, and all the iterations are independent. This information is not necessary to execute the particular program but comes in handy when we transform the program. In this case, we will know that we can safely parallelize or reorder loops related to `vi` as long as we visit all the index elements from `0` to `128`.

2.3 Summary

- Primitive tensor function refers to the single unit of computation in model execution.
  - A MLC process can choose to transform implementation of primitive tensor functions.
- Tensor program is an effective abstraction to represent primitive tensor functions.
  - Key elements include: multi-dimensional buffer, loop nests, computation statement.
  - Program-based transformations can be used to optimize tensor programs.
  - Extra structure can help to provide more information to the transformations.

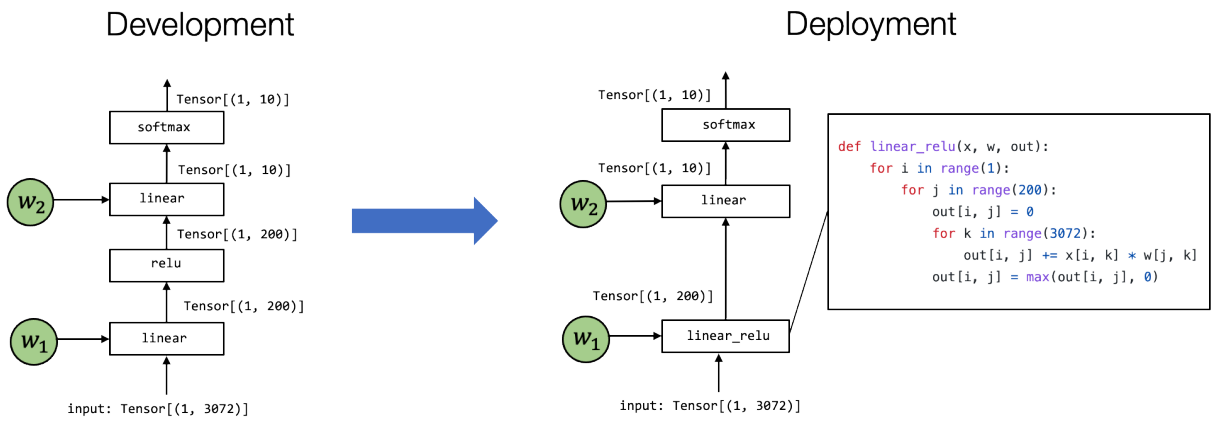
2.4 TensorIR: Tensor Program Abstraction Case Study

2.4.1 Install Packages

For the purpose of this course, we will use some on-going development in tvm, which is an open source machine learning compilation framework. We provide the following command to install a packaged version for mlc course.

```
python3 -m  pip install mlc-ai-nightly -f https://mlc.ai/wheels
```

2.4.2 Prelude



Most MLC process can be viewed as transformation among tensor functions (that can be represented with different abstractions).

To begin today's lecture, let us recap the key principle of the MLC process. Most of the MLC process can be viewed as transformation among tensor functions. The main thing we aim to answer in our following up are:

- What are the possible abstractions to represent the tensor function.
- What are possible transformations among the tensor functions.

Today we are going to cover part of that by focusing on primitive tensor functions.

2.4.3 Learning one Tensor Program Abstraction – TensorIR

We have gone over the primitive tensor function and discussed the high-level idea of tensor program abstractions.

Now we are ready to learn one specific instance of tensor program abstraction called TensorIR. TensorIR is the tensor program abstraction in Apache TVM, which is one of the standard machine learning compilation frameworks.

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

The primary purpose of tensor program abstraction is to represent loops and corresponding hardware acceleration choices such as threading, use of specialized hardware instructions, and memory access.

To help our explanations, let us use the following sequence of tensor computations as a motivating example.

Specifically, for two 128×128 matrices A and B, let us perform the following two steps of tensor computations.

- $Y_{i,j} = \sum_k A_{i,k} \times B_{k,j}$
- $C_{i,j} = \diagdown\!\!\lessdot\!\!\approx(Y_{i,j}) = \gtrdot\partial\curvearrowleft(Y_{i,j}, 0)$

The above computations resemble a typical primitive tensor function commonly seen in neural networks – a linear layer with relu activation. To begin with, we can implement the two operations using array computations in NumPy as follows.

```
dtype = "float32"
a_np = np.random.rand(128, 128).astype(dtype)
b_np = np.random.rand(128, 128).astype(dtype)
# a @ b is equivalent to np.matmul(a, b)
c_mm_relu = np.maximum(a_np @ b_np, 0)
```

Under the hood, NumPy calls into libraries (such as OpenBLAS) and some of its own implementations in lower-level C languages to execute these computations.

From the tensor program abstraction point of view, we would like to see through the details **under the hood** of these array computations. Specifically, we want to ask: what are the possible ways to implement the corresponding computations?

For the purpose of illustrating details under the hood, we will write examples in a restricted subset of NumPy API – which we call **low-level numpy** that uses the following conventions:

- We will use a loop instead of array functions when necessary to demonstrate the possible loop computations.
- When possible, we always explicitly allocate arrays via numpy.empty and pass them around.

Note that this is not how one would typically write NumPy programs. Still, they closely resemble what happens under the hood – most real-world deployment solutions handle allocations separately from computations. The specific libraries perform the computation using different forms of loops and arithmetic computations. Of course, primarily, they are implemented using lower-level languages such as `C`.

```
def lnumpy_mm_relu(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    Y = np.empty((128, 128), dtype="float32")
    for i in range(128):
        for j in range(128):
            for k in range(128):
                if k == 0:
                    Y[i, j] = 0
                Y[i, j] = Y[i, j] + A[i, k] * B[k, j]
    for i in range(128):
        for j in range(128):
            C[i, j] = max(Y[i, j], 0)
```

The program above is one way to implement the `mm_relu` operation. The program contains two stages: first we allocate an intermediate storage Y and store the result of matrix multiplication there. Then we compute the relu in a second sequence of for loops. One thing you might notice is that this is certainly not the only way to implement the `mm_relu`. Likely this is also not the first thing that you might come up with on top of your mind.

Nevertheless, this is one way to implement `mm_relu`, we can verify the correctness of the code by comparing our result to the original one using array computation. We will come back and revisit other possible ways in the later part of this tutorial.

```
c_np = np.empty((128, 128), dtype=dtype)
lnumpy_mm_relu(a_np, b_np, c_np)
np.testing.assert_allclose(c_mm_relu, c_np, rtol=1e-5)
```

The above example code shows how we can bring an **under the hood** implementation of `mm_relu`. Of course, the code itself will run much slower because of the python interpreter. Nevertheless, the example numpy code contains all the possible elements we will use in real-world implementations of those computations.

- Multi-dimensional buffer (arrays).
- Loops over array dimensions.
- Computations statements are executed under the loops.

With the low-level NumPy example in mind, now we are ready to introduce TensorIR. The code block below shows a TensorIR implementation of `mm_relu`. The particular code is implemented in a language called TVMScript, which is a domain-specific dialect embedded in python AST.

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def mm_relu(A: T.Buffer((128, 128), "float32"),
                B: T.Buffer((128, 128), "float32"),
                C: T.Buffer((128, 128), "float32")):
        T.func_attr({"global_symbol": "mm_relu", "tir.noalias": True})
        Y = T.alloc_buffer((128, 128), dtype="float32")
        for i, j, k in T.grid(128, 128, 128):
            with T.block("Y"):
                vi = T.axis.spatial(128, i)
                vj = T.axis.spatial(128, j)
                vk = T.axis.reduce(128, k)
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
        for i, j in T.grid(128, 128):
            with T.block("C"):
                vi = T.axis.spatial(128, i)
                vj = T.axis.spatial(128, j)
                C[vi, vj] = T.max(Y[vi, vj], T.float32(0))
```

It is helpful to be able to see the numpy code and the TensorIR code side-by-side and check the corresponding elements, and we are going to walk through each of them in detail.

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def mm_relu(A: T.Buffer[(128, 128), "float32"],
                B: T.Buffer[(128, 128), "float32"],
                C: T.Buffer[(128, 128), "float32"]):
        T.func_attr({"global_symbol": "mm_relu", "tir.noalias": True})
        Y = T.alloc_buffer((128, 128), dtype="float32")
        for i, j, k in T.grid(128, 128, 128):
            with T.block("Y"):
                vi = T.axis.spatial(128, i)
                vj = T.axis.spatial(128, j)
                vk = T.axis.reduce(128, k)
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
        for i, j in T.grid(128, 128):
            with T.block("C"):
                vi = T.axis.spatial(128, i)
                vj = T.axis.spatial(128, j)
                C[vi, vj] = T.max(Y[vi, vj], T.float32(0))
```

```
def numpy_mm_relu(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    Y = np.empty((128, 128), dtype="float32")
    for i in range(128):
        for j in range(128):
            for k in range(128):
                if k == 0:
                    Y[i, j] = 0
                Y[i, j] = Y[i, j] + A[i, k] * B[k, j]
    for i in range(128):
        for j in range(128):
            C[i, j] = max(Y[i, j], 0)
```

Let us first start by reviewing elements that have a direct correspondence between the numpy and TensorIR side. Then we will come back and review additional elements that are not part of the numpy program.

Function Parameters and Buffers

First, let us see the function parameters. The function parameters correspond to the same set of parameters on the numpy function.

```
# TensorIR
def mm_relu(A: T.Buffer[(128, 128), "float32"],
            B: T.Buffer[(128, 128), "float32"],
            C: T.Buffer[(128, 128), "float32"]):
    ...
# numpy
def lnumpy_mm_relu(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    ...
```

Here A, B, and C takes a type named `T.Buffer`, which with shape argument `(128, 128)` and data type `float32`. This additional information helps possible MLC process to generate code that specializes in the shape and data type.

Similarly, TensorIR also uses a buffer type in intermediate result allocation.

```
# TensorIR
Y = T.alloc_buffer((128, 128), dtype="float32")
# numpy
Y = np.empty((128, 128), dtype="float32")
```

For Loop Iterations

There are also direct correspondence of loop iterations. `T.grid` is a syntactic sugar in TensorIR for us to write multiple nested iterators.

```
# TensorIR
for i, j, k in T.grid(128, 128, 128):

# numpy
for i in range(128):
    for j in range(128):
        for k in range(128):
```

Computational Block

One of the main differences comes from the computational statement. TensorIR contains an additional construct called `T.block`.

```
# TensorIR
with T.block("Y"):
    vi = T.axis.spatial(128, i)
    vj = T.axis.spatial(128, j)
    vk = T.axis.reduce(128, k)
    with T.init():
        Y[vi, vj] = T.float32(0)
    Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
```

(continues on next page)

(continued from previous page)

```
# coressponding numpy code
vi, vj, vk = i, j, k
if vk == 0:
    Y[vi, vj] = 0
Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
```

A **block** is a basic unit of computation in TensorIR. Notably, the block contains a few additional pieces of information compared to the plain NumPy code. A block contains a set of block axes (`vi, vj, vk`) and computations defined around them.

```
vi = T.axis.spatial(128, i)
vj = T.axis.spatial(128, j)
vk = T.axis.reduce(128, k)
```

The above three lines declare the **key properties** about block axes in the following syntax.

```
[block_axis] = T.axis.[axis_type]([axis_range], [mapped_value])
```

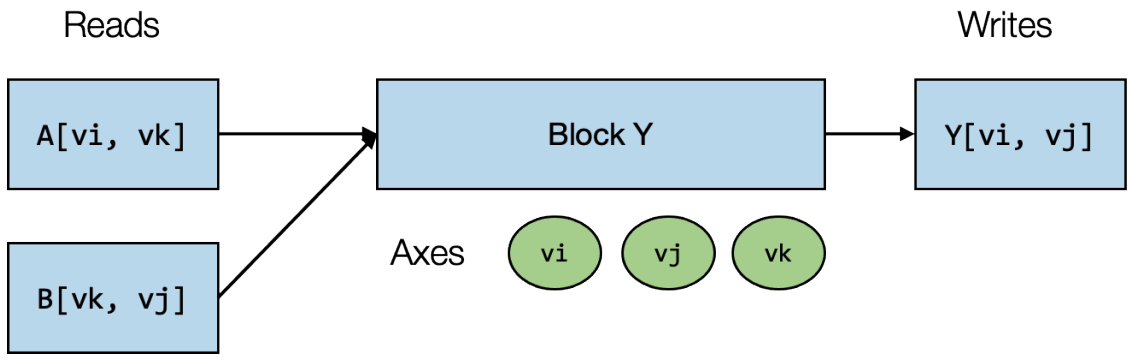
The three lines contain the following information:

- They define where should vi, vj, vk be bound to (in this case i, j k).
- They declare the original range that the vi, vj, vk are supposed to be (the `128` in `T.axis.spatial(128, i)`)
- They declare the properties of the iterators (`spatial`, `reduce`)

Let us walk through those property one by one. First of all, in terms of the bounding relation. `vi = T.axis.spatial(128, i)` effectively implies `vi = i`. The `[axis_range]` value provided the expected range of the `[block_axis]`. For example, `128` in `vi = T.axis.spatial(128, i)` provides an indication that `vi` should be in the `range(0, 128)`.

Block Axis Properties

Let us now start to take a closer look at the block axis properties. These axis properties marks the relation of the axis to the computation being performed. The figure below summarizes the block (iteration) axes and the read write relations of block Y. Note that strictly speaking the block is doing (reduction) updates to `Y`, we mark this as write for now as we don't need value of `Y` from another block.



In our example, block Y computes the result `Y[vi, vj]` by reading values from `A[vi, vk]` and `B[vk, vj]` and perform sum over all possible `vk`. In this particular example, if we fix `vi`, `vj` to be `(0, 1)`, and run the block for `vk in range(0, 128)`, we can effectively compute `C[0, 1]` independently from other possible locations (that have different values of vi, vj).

Notably, for a fixed value of vi and vj, the computation block produces a point value at a spatial location of Y (`Y[vi, vj]`) that is independent from other locations in `Y` (with a different `vi, vj` values). We can call `vi, vj` **spatial axes** as they directly correspond to the beginning of a spatial region of buffers that the block writes to. The axes involved in reduction (`vk`) are named **reduce axes**.

Why Extra Information in Block

One crucial observation is that the additional information (block axis range and their properties) makes the block **self-contained** when it comes to the iterations that it is supposed to carry out independently from the external loop-nest `i, j, k`.

The block axis information also provides additional properties that help us validate the correctness of the external loops used to carry out the computation. For example, the code block below will result in an error because the loop expects an iterator of size `128`, but we only bound it to a for loop of size `127`.

```
# wrong program due to loop and block iteration mismatch
for i in range(127):
    with T.block("C"):
        vi = T.axis.spatial(128, i)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^
        error here due to iterator size mismatch
        ...
```

This additional information also helps us in following machine learning compilation analysis. For example, while we can always parallelize over spatial axes, parallelizing over reduce axes will require specific strategies.

Sugar for Block Axes Binding

In situations where each of the block axes is directly mapped to an outer loop iterator, we can use `T.axis.remap` to declare the block axis in a single line.

```
# SSR means the properties of each axes are "spatial", "spatial", "reduce"
vi, vj, vk = T.axis.remap("SSR", [i, j, k])
```

is equivalent to

```
vi = T.axis.spatial(range_of_i, i)
vj = T.axis.spatial(range_of_j, j)
vk = T.axis.reduce(range_of_k, k)
```

So we can also write the programs as follows.

```
@tvm.script.ir_module
class MyModuleWithAxisRemapSugar:
    @T.prim_func
    def mm_relu(A: T.Buffer((128, 128), "float32"),
```

(continues on next page)

(continued from previous page)

```
            B: T.Buffer((128, 128), "float32"),
            C: T.Buffer((128, 128), "float32")):
        T.func_attr({"global_symbol": "mm_relu", "tir.noalias": True})
        Y = T.alloc_buffer((128, 128), dtype="float32")
        for i, j, k in T.grid(128, 128, 128):
            with T.block("Y"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
        for i, j in T.grid(128, 128):
            with T.block("C"):
                vi, vj = T.axis.remap("SS", [i, j])
                C[vi, vj] = T.max(Y[vi, vj], T.float32(0))
```

Function Attributes and Decorators

So far, we have covered most of the elements in TensorIR. In this part, we will go over the remaining elements of the script.

The function attribute information contains extra information about the function.

```
T.func_attr({"global_symbol": "mm_relu", "tir.noalias": True})
```

Here `global_symbol` corresponds to the name of the function, and `tir.noalias` is an attribute indicating that all the buffer memory areas do not overlap. You also feel free safely skip these attributes for now as they won't affect the overall understanding of the high-level concepts.

The two decorators, `@tvm.script.ir_module` and `@T.prim_func` are used to indicate the type of the corresponding part.

`@tvm.script.ir_module` indicates that MyModule is an `IRModule`. IRModule is the container object to hold a collection of tensor functions in machine learning compilation.

```
type(MyModule)
```

```
tvm.ir.module.IRModule
```

```
type(MyModule["mm_relu"])
```

```
tvm.tir.function.PrimFunc
```

Up until now, we have only seen IRModules containing a single tensor function. An IRModule in the MLC process can contain multiple tensor functions. The following code block shows an example of an IRModule with two functions.

```
@tvm.script.ir_module
class MyModuleWithTwoFunctions:
    @T.prim_func
```

(continues on next page)

(continued from previous page)

```
    def mm(A: T.Buffer((128, 128), "float32"),
           B: T.Buffer((128, 128), "float32"),
           Y: T.Buffer((128, 128), "float32")):
        T.func_attr({"global_symbol": "mm", "tir.noalias": True})
        for i, j, k in T.grid(128, 128, 128):
            with T.block("Y"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]

    @T.prim_func
    def relu(A: T.Buffer((128, 128), "float32"),
             B: T.Buffer((128, 128), "float32")):
        T.func_attr({"global_symbol": "relu", "tir.noalias": True})
        for i, j in T.grid(128, 128):
            with T.block("B"):
                vi, vj = T.axis.remap("SS", [i, j])
                B[vi, vj] = T.max(A[vi, vj], T.float32(0))
```

Section Checkpoint

So far, we have gone through one example instance of TensorIR program and covered most of the elements, including:

- Buffer declarations in parameters and intermediate temporary memory.
- For loop iterations.
- **Blocks** and block axes properties.

In this section, we have gone through one example instance of TensorIR that covers the most common elements in MLC.

TensorIR contains more elements than what we went over in this section, but this section covers most of the key parts that can get us started in the MLC journey. We will cover new elements as we encounter them in the later chapters.

2.4.4 Transformation

In the last section, we learned about TensorIR and its key elements. Now, let us get to the main ingredients of all MLC flows – transformations of primitive tensor functions.

In the last section, we have given an example of how to write `mm_relu` using low-level numpy. In practice, there can be multiple ways to implement the same functionality, and each implementation can result in different performance.

We will discuss the reason behind the performance difference and how to leverage those variants in future lectures. In this lecture, let us focus on the ability to get different implementation variants using transformations.

```
def lnumpy_mm_relu_v2(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    Y = np.empty((128, 128), dtype="float32")
```

(continues on next page)

(continued from previous page)

```
    for i in range(128):
        for j0 in range(32):
            for k in range(128):
                for j1 in range(4):
                    j = j0 * 4 + j1
                    if k == 0:
                        Y[i, j] = 0
                    Y[i, j] = Y[i, j] + A[i, k] * B[k, j]
    for i in range(128):
        for j in range(128):
            C[i, j] = max(Y[i, j], 0)

c_np = np.empty((128, 128), dtype=dtype)
lnumpy_mm_relu_v2(a_np, b_np, c_np)
np.testing.assert_allclose(c_mm_relu, c_np, rtol=1e-5)
```

The above code block shows a slightly different variation of `mm_relu`. To see the relation to the original program

- We replace the `j` loop with two loops, `j0` and `j1`.
- The order of iterations changes slightly

In order to get `lnumpy_mm_relu_v2`, we have to rewrite it into a new function (or manually copy-paste and edit). TensorIR introduces a utility called Schedule that allows us to do that pragmatically.

To remind ourselves, let us look again at the current MyModule content.

```
import IPython

IPython.display.Code(MyModule.script(), language="python")
```

Now we are ready to try out the code transformations, we begin by creating a `Schedule` helper class with the given MyModule as input.

```
sch = tvm.tir.Schedule(MyModuleWithAxisRemapSugar)
```

Then we perform the following operations to obtain a reference to block Y and corresponding loops.

```
block_Y = sch.get_block("Y", func_name="mm_relu")
i, j, k = sch.get_loops(block_Y)
```

Now we are ready to perform the transformations. The first transformation we will perform is to split the loop `j` into two loops, with the length of the inner loop to be `4`. Note that the transformation is procedural, so if you accidentally execute the block twice, we will get an error that variable `j` no longer exists. If that happens, you can run again from the beginning (where `sch` get created).

```
j0, j1 = sch.split(j, factors=[None, 4])
```

We can look at the result of the transformation, which is stored at `sch.mod`.

```
IPython.display.Code(sch.mod.script(), language="python")
```

After the first step of transformation, we created two additional loops, `j_0` and `j_1`, with corresponding ranges 32 and 4. Our next step would be to reorder the two loops.

```
sch.reorder(j0, k, j1)
IPython.display.Code(sch.mod.script(), language="python")
```

Now code after reordering closely resembles `lnumpy_mm_relu_v2`.

Getting to Another Variant

In this section, we are going to go ahead and do another two steps of transformations to get to another variant. First, we use a primitive called `reverse_compute_at` to move block C to an inner loop of `Y`.

```
block_C = sch.get_block("C", "mm_relu")
sch.reverse_compute_at(block_C, j0)
IPython.display.Code(sch.mod.script(), language="python")
```

So far, we have kept the reduction initialization and update step together in a single block body. This combined form brings convenience for loop transformations (as outer loop i,j of initialization and updates usually need to keep in sync with each other).

After loop transformations, we can move the initialization of `Y`'s element separate from the reduction update. We can do that through the `decompose_reduction` primitive. (note: this is also done implicitly by tvm during future compilation, so this step is mainly to make it explicit and see the end effect).

```
sch.decompose_reduction(block_Y, k)
IPython.display.Code(sch.mod.script(), language="python")
```

The final transformed code resembles the following low-level NumPy code.

```
def lnumpy_mm_relu_v3(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    Y = np.empty((128, 128), dtype="float32")
    for i in range(128):
        for j0 in range(32):
            # Y_init
            for j1 in range(4):
                j = j0 * 4 + j1
                Y[i, j] = 0
            # Y_update
            for k in range(128):
                for j1 in range(4):
                    j = j0 * 4 + j1
                    Y[i, j] = Y[i, j] + A[i, k] * B[k, j]
            # C
            for j1 in range(4):
                j = j0 * 4 + j1
                C[i, j] = max(Y[i, j], 0)

c_np = np.empty((128, 128), dtype=dtype)
lnumpy_mm_relu_v3(a_np, b_np, c_np)
np.testing.assert_allclose(c_mm_relu, c_np, rtol=1e-5)
```

Section Summary and Discussions

The main takeaway of this section is to get used to the paradigm of incremental code transformations. In our particular example, we use `tir.Schedule` as an auxiliary helper object.

Importantly, we avoided the need to re-create different variants of the same program (`lnumpy_mm_relu`, `lnumpy_mm_relu_v2` and `lnumpy_mm_relu_v3`). The additional information in blocks (axes information) is the reason we can do such transformations under the hood.

2.4.5 Build and Run

So far, we have only looked at the script output of the transformed result. We can also run the program obtained in IRModule.

First, we call a build function to turn an IRModule into a `runtime.Module`, representing a collection of runnable functions. Here target specifies detailed information about the deployment environment. For this particular case, we will use `llvm`, which helps us compile to the native CPU platform.

When we target different platforms(e.g. an Android phone) or platforms with special instructions (intel skylake), we will need to adjust the target accordingly. We will discuss different target choices as we start to deploy to those environments.

```
rt_lib = tvm.build(MyModule, target="llvm")
```

Then, we will create three tvm ndarrays that are used to hold inputs and the output.

```
a_nd = tvm.nd.array(a_np)
b_nd = tvm.nd.array(b_np)
c_nd = tvm.nd.empty((128, 128), dtype="float32")
type(c_nd)
```

```
tvm.runtime.ndarray.NDArray
```

Finally, we can get the runnable function from rt_lib and execute it by passing the three array arguments. We can further run validation to check the code difference.

```
func_mm_relu = rt_lib["mm_relu"]
func_mm_relu(a_nd, b_nd, c_nd)

np.testing.assert_allclose(c_mm_relu, c_nd.numpy(), rtol=1e-5)
```

We have built and run the original MyModule. We can also build the transformed program.

```
rt_lib_after = tvm.build(sch.mod, target="llvm")
rt_lib_after["mm_relu"](a_nd, b_nd, c_nd)
np.testing.assert_allclose(c_mm_relu, c_nd.numpy(), rtol=1e-5)
```

Finally, we can compare the time difference between the two. `time_evaluator` is a helper benchmarking function that can be used to compare the running performance of different generated functions.

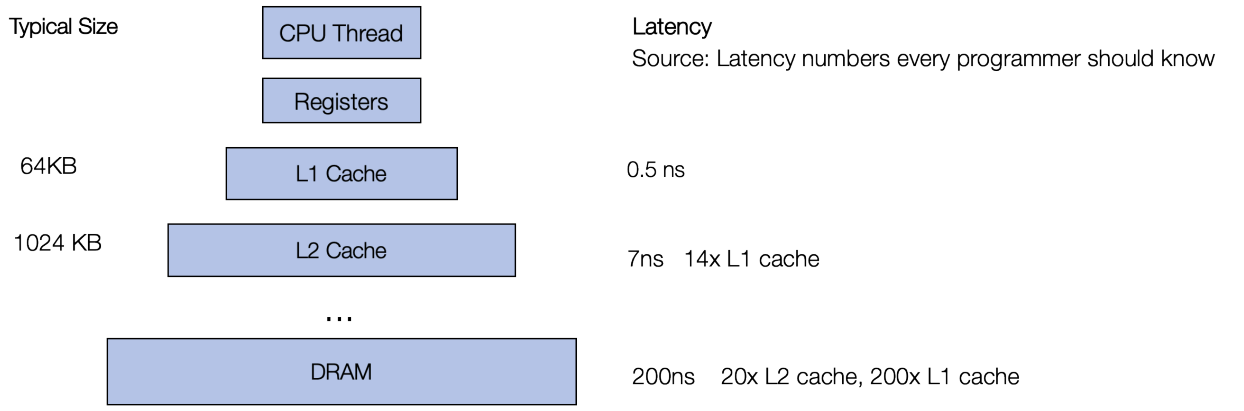
```
f_timer_before = rt_lib.time_evaluator("mm_relu", tvm.cpu())
print("Time cost of MyModule %g sec" % f_timer_before(a_nd, b_nd, c_nd).mean)
f_timer_after = rt_lib_after.time_evaluator("mm_relu", tvm.cpu())
print("Time cost of transformed sch.mod %g sec" % f_timer_after(a_nd, b_nd, c_
↪nd).mean)
```

```
Time cost of MyModule 0.000891969 sec
Time cost of transformed sch.mod 0.000273506 sec
```

It is interesting to see the running time difference between the two codes. Let us do a quick analysis of what are the possible factors that affect the performance. First, let us remind ourselves of two variants of the code.

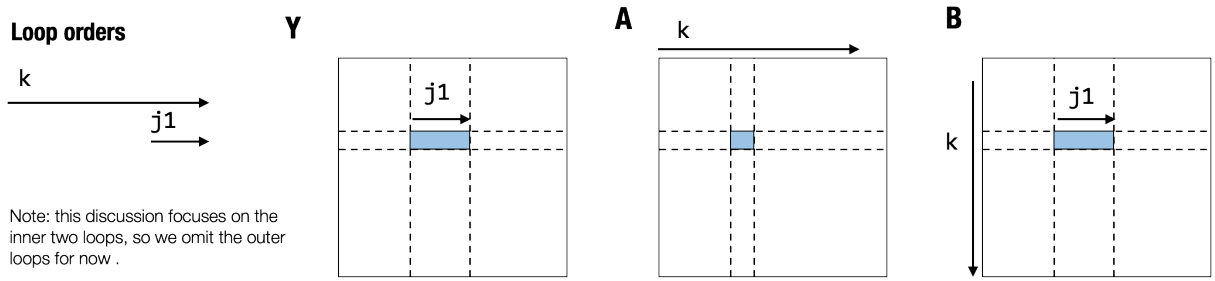
```
import IPython

IPython.display.Code(MyModule.script(), language="python")
```

```
IPython.display.Code(sch.mod.script(), language="python")
```



To see why different loop variants result in different performances, we need to review the fact that it is not uniformly fast to access any piece of memory in `A` and `B`. Modern CPU comes with multiple levels of caches, where data needs to be fetched into the cache before the CPU can access it.

Importantly, it is much faster to access the data already in the cache. One strategy that CPU takes is to fetch data closer to each other. When we read one element in the memory, it will attempt to fetch the elements close by (formally known as cache-line) to the cache. So when you read the next element, it is already in the cache. As a result, code with continuous memory access is usually faster than code that randomly accesses different parts of the memory.



Now let us look at the above visualization of iterations and analyze what is going on. In this analysis, let us focus on two inner-most loops: `k` and `j1`. The highlighted cover shows the corresponding region in `Y`, `A` and `B` that the iteration touches when we iterate over `j1` for one specific instance of `k`.

We can find that the `j1` iteration produces **continuous access** to elements of `B`. Specifically, it means the values we read when `j1=0` and `j1=1` are next to each other. This enables better cache access behavior. In addition, we bring the computation of C closer to `Y`, enabling better caching behavior.

Our current example is mainly to demonstrate that different variants of code can lead to different performances. More transformation steps can help us to get to even better performance, which we will cover in future chapters. The main goal of this exercise is first to get us the tool of program transformations and first taste of what is possible through transformations.

Exercise

As an exercise, try different `j_factor` choices and see how they affect the code's performance.

```
def transform(mod, jfactor):
    sch = tvm.tir.Schedule(mod)
    block_Y = sch.get_block("Y", func_name="mm_relu")
    i, j, k = sch.get_loops(block_Y)
    j0, j1 = sch.split(j, factors=[None, jfactor])
    sch.reorder(j0, k, j1)
    block_C = sch.get_block("C", "mm_relu")
    sch.reverse_compute_at(block_C, j0)
    return sch.mod

mod_transformed = transform(MyModule, jfactor=8)

rt_lib_transformed = tvm.build(mod_transformed, "llvm")
f_timer_transformed = rt_lib_transformed.time_evaluator("mm_relu", tvm.cpu())
print("Time cost of transformed mod_transformed %g sec" % f_timer_
↪transformed(a_nd, b_nd, c_nd).mean)
# display the code below
IPython.display.Code(mod_transformed.script(), language="python")
```

```
Time cost of transformed mod_transformed 0.000159384 sec
```

2.4.6 Ways to Create and Interact with TensorIR

In the last sections, we learned about TensorIR abstraction and ways to transform things. TensorIR comes with an additional construct named block that helps us analyze and perform code transformations. One natural question we might ask: what are common ways to create and interact with TensorIR functions?

Create TensorIR via TVMScript

The first way to get a TensorIR function is to write a function in TVMScript directly, and this is also the approach we use in the last sections. TVMScript also allows us to skip certain parts of information when necessary. For example, `T.axis.remap` enables us to shorten the iterator size annotations.

TVMScript is also a useful way to inspect the tensor functions in the middle of transformations. In some instances, it might be helpful to print out the script, do some manual editing, then feed it back to the MLC process just to debug and try out possible transformation (manually), then bake it into the MLC process.

Generate TensorIR code using Tensor Expression

In many cases, our development forms are higher-level abstractions that are not at the loop level. So another common way to obtain TensorIR is programmatically generating relevant code.

Tensor expression (te) is a domain-specific language that describes a sequence of computations via an expression-like API.

```
from tvm import te

A = te.placeholder((128, 128), "float32", name="A")
B = te.placeholder((128, 128), "float32", name="B")
k = te.reduce_axis((0, 128), "k")
Y = te.compute((128, 128), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),␣
↪name="Y")
C = te.compute((128, 128), lambda i, j: te.max(Y[i, j], 0), name="C")
```

Here `te.compute` takes the signature `te.compute(output_shape, fcompute)`. And the fcompute function describes how we want to compute the value of each element `Y[i, j]` for a given index.

```
lambda i, j: te.sum(A[i, k] * B[k, j], axis=k)
```

The above lambda expression describes the computation $Y_{ij} = \sum_k A_{ik} B_{kj}$. After describing the computation, we can create a TensorIR function by passing the relevant parameter we are interested in. In this particular case, we want to create a function with two input parameters (A, B) and one output parameter (C).

```
te_func = te.create_prim_func([A, B, C]).with_attr({"global_symbol": "mm_relu
↪"})
MyModuleFromTE = tvm.IRModule({"mm_relu": te_func})
IPython.display.Code(MyModuleFromTE.script(), language="python")
```

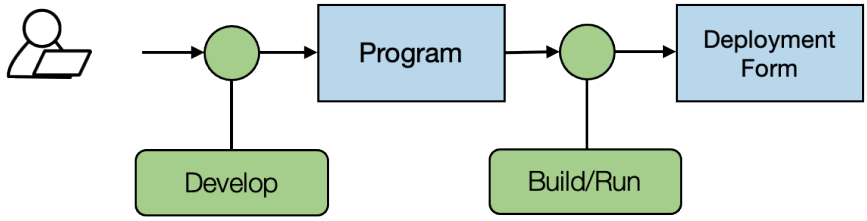
The tensor expression API provides a helpful tool to generate TensorIR functions for a given higher-level input.

2.4.7 TensorIR Functions as Results of Transformations

In practice, we also get TensorIR functions as results of transformations. This happens when we start with two primitive tensor functions (mm and relu), then apply a programmatic transformation to "fuse" them into a single primitive tensor function, `mm_relu`. We will cover the details in future chapters.

2.4.8 Discussions

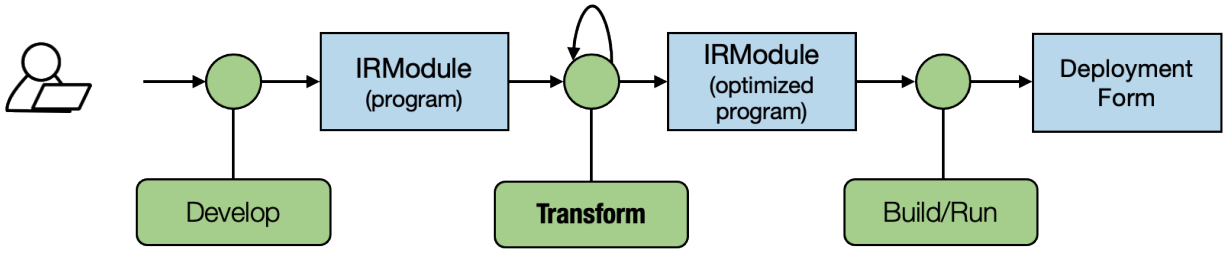
In this section, let us review what we learned so far. We learned that a common MLC process follows a sequence of program transformations. It is interesting to compare the TensorIR transformation process to the low-level numpy reference development process.



The above figure shows the standard development process. We need to repeat the process of developing different program variants and then (build if it is a compiled language) run them on the platform of interest.

The key difference in an MLC process(shown in the figure below) is the programmatic transformations among the IRModule (programs). So we can not only come up with program variants through development (either by manually writing the code or generating the code), but also can obtain variants by transforming the tensor programs.

Transformation is a very powerful tool that helps us simplify development costs and introduce more automation to the process. This section covered a specific perspective on primitive tensor functions via TensorIR, and we will cover more perspectives in the future.



Notably, direct code development and transformations are equally important in practice: We can still leverage a lot of domain expertise to develop and optimize part of the programs and then combine that with transformation-based approaches. We will talk about how to combine the two practices in future chapters.

2.4.9 Summary

- TensorIR abstraction
  - Contains common elements such as loops, multi-dimensional buffers
  - Introduced a new construct **Block** that encapsulates the loop computation requirements.
  - Can be constructed in python AST(via TVMScript)
- We can use transformations to create different variants of TensorIR.
- Common MLC flow: develop, transform, build.

2.5 Exercises for TensorIR

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

2.5.1 Section 1: How to Write TensorIR

In this section, let's try to write TensorIR manually according to high-level instructions (e.g., Numpy or Torch). First, we give an example of element-wise add function, to show what should we do to write a TensorIR function.

Example: Element-wise Add

First, let's try to use Numpy to write an element-wise add function.

```
# init data
a = np.arange(16).reshape(4, 4)
b = np.arange(16, 0, -1).reshape(4, 4)
```

```
# numpy version
c_np = a + b
c_np
```

```
array([[16, 16, 16, 16],
       [16, 16, 16, 16],
       [16, 16, 16, 16],
       [16, 16, 16, 16]])
```

Before we directly write TensorIR, we should first translate high-level computation abstraction (e.g., `ndarray + ndarray`) to low-level python implementation (standard for loops with element access and operation)

Notably, the initial value of the output array (or buffer) is not always `0`. We need to write or initialize it in our implementation, which is important for reduction operator (e.g. matmul and conv)

```
# low-level numpy version
def lnumpy_add(a: np.ndarray, b: np.ndarray, c: np.ndarray):
  for i in range(4):
    for j in range(4):
      c[i, j] = a[i, j] + b[i, j]
c_lnumpy = np.empty((4, 4), dtype=np.int64)
lnumpy_add(a, b, c_lnumpy)
c_lnumpy
```

```
array([[16, 16, 16, 16],
       [16, 16, 16, 16],
       [16, 16, 16, 16],
       [16, 16, 16, 16]])
```

Now, let's take a further step: translate low-level NumPy implementation into TensorIR. And compare the result with it comes from NumPy.

```
# TensorIR version
@tvm.script.ir_module
class MyAdd:
  @T.prim_func
  def add(A: T.Buffer((4, 4), "int64"),
          B: T.Buffer((4, 4), "int64"),
          C: T.Buffer((4, 4), "int64")):
    T.func_attr({"global_symbol": "add"})
    for i, j in T.grid(4, 4):
      with T.block("C"):
        vi = T.axis.spatial(4, i)
        vj = T.axis.spatial(4, j)
        C[vi, vj] = A[vi, vj] + B[vi, vj]

rt_lib = tvm.build(MyAdd, target="llvm")
a_tvm = tvm.nd.array(a)
b_tvm = tvm.nd.array(b)
c_tvm = tvm.nd.array(np.empty((4, 4), dtype=np.int64))
rt_lib["add"](a_tvm, b_tvm, c_tvm)
np.testing.assert_allclose(c_tvm.numpy(), c_np, rtol=1e-5)
```

Here, we have finished the TensorIR function. Please take your time to finish the following exercises

Exercise 1: Broadcast Add

Please write a TensorIR function that adds two arrays with broadcasting.

```
# init data
a = np.arange(16).reshape(4, 4)
b = np.arange(4, 0, -1).reshape(4)
```

```
# numpy version
c_np = a + b
c_np
```

```
array([[ 4,  4,  4,  4],
       [ 8,  8,  8,  8],
       [12, 12, 12, 12],
       [16, 16, 16, 16]])
```

Please complete the following Module `MyAdd` and run the code to check your implementation.

```
@tvm.script.ir_module
class MyAdd:
  @T.prim_func
  def add():
    T.func_attr({"global_symbol": "add", "tir.noalias": True})
    # TODO
    ...

rt_lib = tvm.build(MyAdd, target="llvm")
a_tvm = tvm.nd.array(a)
b_tvm = tvm.nd.array(b)
c_tvm = tvm.nd.array(np.empty((4, 4), dtype=np.int64))
rt_lib["add"](a_tvm, b_tvm, c_tvm)
np.testing.assert_allclose(c_tvm.numpy(), c_np, rtol=1e-5)
```

Exercise 2: 2D Convolution

Then, let's try to do something challenging: 2D convolution, which is a common operation in image processing.

Here is the mathematical definition of convolution with NCHW layout:

$$Conv[b,k,i,j] = \sum_{di,dj,q} A[b,q,strides*i+di,strides*j+dj]*W[k,q,di,dj] \tag{2.5.1}$$

, where, `A` is the input tensor, `W` is the weight tensor, `b` is the batch index, `k` is the out channels, `i` and `j` are indices for image hight and width, `di` and `dj` are the indices of the weight, `q` is the input channel, and `strides` is the stride of the filter window.

In the exercise, we pick a small and simple case with `stride=1, padding=0`.

```
N, CI, H, W, CO, K = 1, 1, 8, 8, 2, 3
OUT_H, OUT_W = H - K + 1, W - K + 1
data = np.arange(N*CI*H*W).reshape(N, CI, H, W)
weight = np.arange(CO*CI*K*K).reshape(CO, CI, K, K)
```

```
# torch version
import torch

data_torch = torch.Tensor(data)
weight_torch = torch.Tensor(weight)
conv_torch = torch.nn.functional.conv2d(data_torch, weight_torch)
conv_torch = conv_torch.numpy().astype(np.int64)
conv_torch
```

```
array([[[[ 474,  510,  546,  582,  618,  654],
         [ 762,  798,  834,  870,  906,  942],
         [1050, 1086, 1122, 1158, 1194, 1230],
         [1338, 1374, 1410, 1446, 1482, 1518],
         [1626, 1662, 1698, 1734, 1770, 1806],
         [1914, 1950, 1986, 2022, 2058, 2094]],

        [[1203, 1320, 1437, 1554, 1671, 1788],
         [2139, 2256, 2373, 2490, 2607, 2724],
         [3075, 3192, 3309, 3426, 3543, 3660],
         [4011, 4128, 4245, 4362, 4479, 4596],
         [4947, 5064, 5181, 5298, 5415, 5532],
         [5883, 6000, 6117, 6234, 6351, 6468]]]])
```

Please complete the following Module `MyConv` and run the code to check your implementation.

```
@tvm.script.ir_module
class MyConv:
  @T.prim_func
  def conv():
    T.func_attr({"global_symbol": "conv", "tir.noalias": True})
    # TODO
    ...

rt_lib = tvm.build(MyConv, target="llvm")
data_tvm = tvm.nd.array(data)
weight_tvm = tvm.nd.array(weight)
conv_tvm = tvm.nd.array(np.empty((N, CO, OUT_H, OUT_W), dtype=np.int64))
rt_lib["conv"](data_tvm, weight_tvm, conv_tvm)
np.testing.assert_allclose(conv_tvm.numpy(), conv_torch, rtol=1e-5)
```

2.5.2 Section 2: How to Transform TensorIR

In the lecture, we learned that TensorIR is not only a programming language but also an abstraction for program transformation. In this section, let's try to transform the program. We take `bmm_relu` (`batched_matmul_relu`) in our studies, which is a variant of operations that common appear in models such as transformers.

Parallel, Vectorize and Unroll

First, we introduce some new primitives, `parallel`, `vectorize` and `unroll`. These three primitives operate on loops to indicate how this loop executes. Here is the example:

```
@tvm.script.ir_module
class MyAdd:
  @T.prim_func
  def add(A: T.Buffer((4, 4), "int64"),
          B: T.Buffer((4, 4), "int64"),
          C: T.Buffer((4, 4), "int64")):
    T.func_attr({"global_symbol": "add"})
    for i, j in T.grid(4, 4):
```

(continues on next page)

(continued from previous page)

```
      with T.block("C"):
        vi = T.axis.spatial(4, i)
        vj = T.axis.spatial(4, j)
        C[vi, vj] = A[vi, vj] + B[vi, vj]

sch = tvm.tir.Schedule(MyAdd)
block = sch.get_block("C", func_name="add")
i, j = sch.get_loops(block)
i0, i1 = sch.split(i, factors=[2, 2])
sch.parallel(i0)
sch.unroll(i1)
sch.vectorize(j)
IPython.display.Code(sch.mod.script(), language="python")
```

Exercise 3: Transform a batch matmul program

Now, let's go back to the `bmm_relu` exercise. First, Let's see the definition of `bmm`:

- $Y_{n,i,j} = \sum_k A_{n,i,k} \times B_{n,k,j}$
- $C_{n,i,j} = \diagdown\!\!<\!\!\approx\!\!(Y_{n,i,j}) = \gtrdot\!\!\supset\!\!\curvearrowleft\!\!(Y_{n,i,j}, 0)$

It's your time to write the TensorIR for `bmm_relu`. We provide the lnumpy func as hint:

```
def lnumpy_mm_relu_v2(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    Y = np.empty((16, 128, 128), dtype="float32")
    for n in range(16):
        for i in range(128):
            for j in range(128):
                for k in range(128):
                    if k == 0:
                        Y[n, i, j] = 0
                    Y[n, i, j] = Y[n, i, j] + A[n, i, k] * B[n, k, j]
    for n in range(16):
        for i in range(128):
            for j in range(128):
                C[n, i, j] = max(Y[n, i, j], 0)
```

```
@tvm.script.ir_module
class MyBmmRelu:
  @T.prim_func
  def bmm_relu():
    T.func_attr({"global_symbol": "bmm_relu", "tir.noalias": True})
    # TODO
    ...

sch = tvm.tir.Schedule(MyBmmRelu)
IPython.display.Code(sch.mod.script(), language="python")
# Also please validate your result
```

In this exercise, let's focus on transform the original program to a specific target. Note that the target program may not be the best one due to different hardware. But this exercise aims to let students understand how to transform the program to a wanted one. Here is the target program:

```
@tvm.script.ir_module
class TargetModule:
    @T.prim_func
    def bmm_relu(A: T.Buffer((16, 128, 128), "float32"), B: T.Buffer((16, 128,
↪ 128), "float32"), C: T.Buffer((16, 128, 128), "float32")) -> None:
        T.func_attr({"global_symbol": "bmm_relu", "tir.noalias": True})
        Y = T.alloc_buffer([16, 128, 128], dtype="float32")
        for i0 in T.parallel(16):
            for i1, i2_0 in T.grid(128, 16):
                for ax0_init in T.vectorized(8):
                    with T.block("Y_init"):
                        n, i = T.axis.remap("SS", [i0, i1])
                        j = T.axis.spatial(128, i2_0 * 8 + ax0_init)
                        Y[n, i, j] = T.float32(0)
                for ax1_0 in T.serial(32):
                    for ax1_1 in T.unroll(4):
                        for ax0 in T.serial(8):
                            with T.block("Y_update"):
                                n, i = T.axis.remap("SS", [i0, i1])
                                j = T.axis.spatial(128, i2_0 * 8 + ax0)
                                k = T.axis.reduce(128, ax1_0 * 4 + ax1_1)
                                Y[n, i, j] = Y[n, i, j] + A[n, i, k] * B[n, k,
↪ j]
                for i2_1 in T.vectorized(8):
                    with T.block("C"):
                        n, i = T.axis.remap("SS", [i0, i1])
                        j = T.axis.spatial(128, i2_0 * 8 + i2_1)
                        C[n, i, j] = T.max(Y[n, i, j], T.float32(0))
```

Your task is to transform the original program to the target program.

```
sch = tvm.tir.Schedule(MyBmmRelu)
# TODO: transformations
# Hints: you can use
# `IPython.display.Code(sch.mod.script(), language="python")`
# or `print(sch.mod.script())`
# to show the current program at any time during the transformation.

# Step 1. Get blocks
Y = sch.get_block("Y", func_name="bmm_relu")
...

# Step 2. Get loops
b, i, j, k = sch.get_loops(Y)
...

# Step 3. Organize the loops
k0, k1 = sch.split(k, ...)
sch.reorder(...)
sch.compute_at/reverse_compute_at(...)
...

# Step 4. decompose reduction
Y_init = sch.decompose_reduction(Y, ...)
...
```

(continues on next page)

(continued from previous page)

```
# Step 5. vectorize / parallel / unroll
sch.vectorize(...)
sch.parallel(...)
sch.unroll(...)
...

IPython.display.Code(sch.mod.script(), language="python")
```

OPTIONAL If we want to make sure the transformed program is exactly the same as the given target, we can use `assert_structural_equal`. Note that this step is an optional step in this exercise. It's good enough if you transformed the program **towards** the target and get performance improvement.

```
tvm.ir.assert_structural_equal(sch.mod, TargetModule)
print("Pass")
```

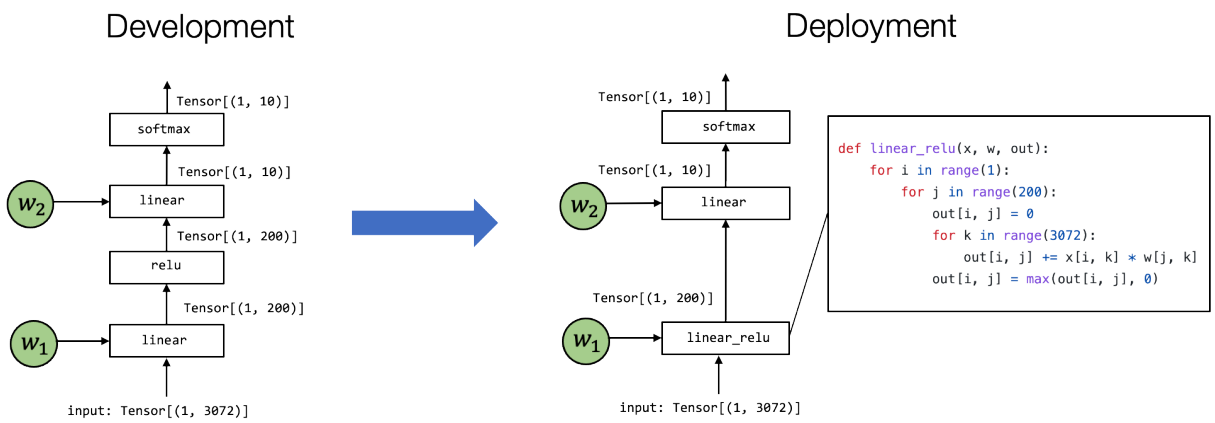
Build and Evaluate

Finally we can evaluate the performance of the transformed program.

```
before_rt_lib = tvm.build(MyBmmRelu, target="llvm")
after_rt_lib = tvm.build(sch.mod, target="llvm")
a_tvm = tvm.nd.array(np.random.rand(16, 128, 128).astype("float32"))
b_tvm = tvm.nd.array(np.random.rand(16, 128, 128).astype("float32"))
c_tvm = tvm.nd.array(np.random.rand(16, 128, 128).astype("float32"))
after_rt_lib["bmm_relu"](a_tvm, b_tvm, c_tvm)
before_timer = before_rt_lib.time_evaluator("bmm_relu", tvm.cpu())
print("Before transformation:")
print(before_timer(a_tvm, b_tvm, c_tvm))

f_timer = after_rt_lib.time_evaluator("bmm_relu", tvm.cpu())
print("After transformation:")
print(f_timer(a_tvm, b_tvm, c_tvm))
```

3 End to End Model Execution

3.1 Prelude



Most MLC process can be viewed as transformation among tensor functions (that can be represented with different abstractions).

Most of the MLC process can be viewed as transformation among tensor functions. The main thing we aim to answer in our following up are:

- What are the possible abstractions to represent the tensor function.
- What are possible transformations among the tensor functions.

In the last lecture, we focus on the primitive tensor functions. In this lecture, we will talk about how to build end-to-end models.

3.2 Preparations

To begin with, we will import necessary dependencies and create helper functions.

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

3.2.1 Load the Dataset

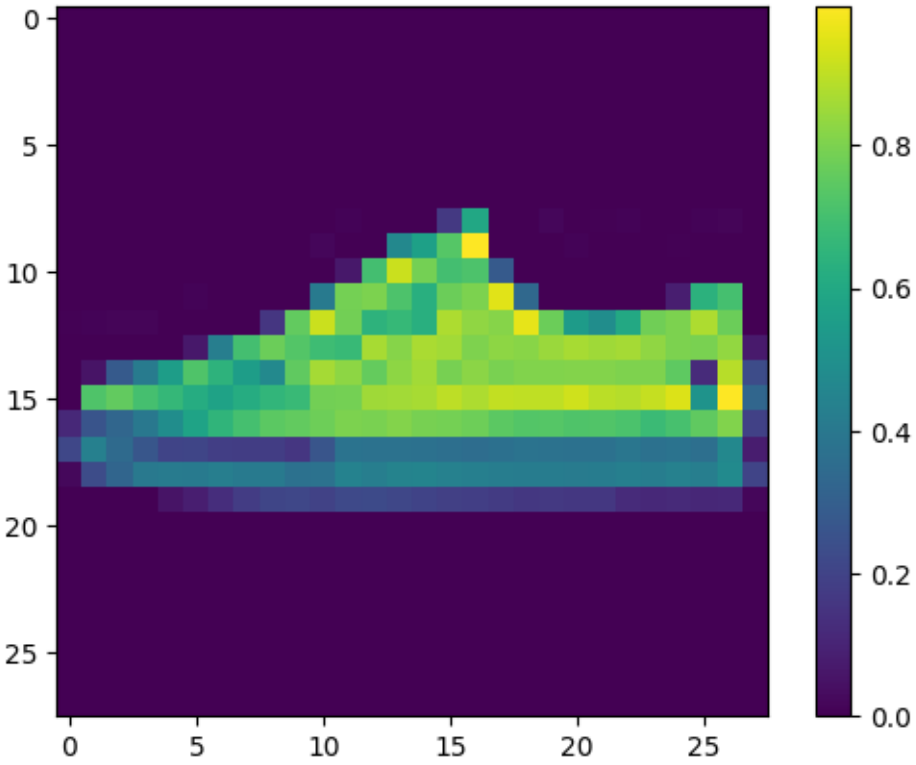
As a concrete example, we will be using a model on the fashion MNIST dataset. The following code downloads and prepares the data from `torchvision` in NumPy array.

```
import torch
import torchvision

test_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=1,␣
↪shuffle=True)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

img, label = next(iter(test_loader))
img = img.reshape(1, 28, 28).numpy()
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100.0%Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw
```

We can plot out the image instance that we want to be able to predict.

```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(img[0])
plt.colorbar()
plt.grid(False)
plt.show()
print("Class:", class_names[label[0]])
```



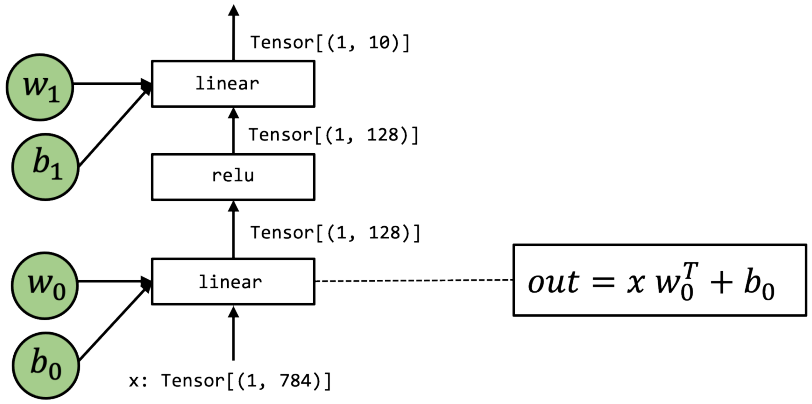

```
Class: Sneaker
```

3.2.2 Download Model Parameters

```
# Hide outputs
!wget https://github.com/mlc-ai/web-data/raw/main/models/fasionmnist_mlp_
↪params.pkl
```

3.3 End to End Model Integration

In this chapter, we will use the following model as an example. This is a two-layer neural network that consists of two linear operations with relu activation. To keep things simple, we removed the final softmax layer. The output score is un-normalized, but still, the maximum value corresponds to the most likely class.



Let us begin by reviewing a Numpy implementation of the model.

```
def numpy_mlp(data, w0, b0, w1, b1):
    lv0 = data @ w0.T + b0
    lv1 = np.maximum(lv0, 0)
    lv2 = lv1 @ w1.T + b1
    return lv2
```

```
import pickle as pkl

mlp_params = pkl.load(open("fasionmnist_mlp_params.pkl", "rb"))
res = numpy_mlp(img.reshape(1, 784),
                mlp_params["w0"],
                mlp_params["b0"],
                mlp_params["w1"],
                mlp_params["b1"])
print(res)
pred_kind = res.argmax(axis=1)
print(pred_kind)
print("NumPy-MLP Prediction:", class_names[pred_kind[0]])
```

```
[[-21.10578   -30.078623  -16.665087  -15.054921  -26.568043   -0.3226527
  -25.358425   20.298119   -2.9159825 -18.293468 ]]
[7]
NumPy-MLP Prediction: Sneaker
```

The above example code shows the high-level array operations to perform the end-to-end model execution.

Again from MLC's pov, we would like to see through the details under the hood of these array computations.

For the purpose of illustrating details under the hood, we will again write examples in low-level numpy:

- We will use a loop instead of array functions when necessary to demonstrate the possible loop computations.

- When possible, we always explicitly allocate arrays via numpy.empty and pass them around.

The code block below shows a low-level numpy implementation of the same model.

```
def lnumpy_linear0(X: np.ndarray, W: np.ndarray, B: np.ndarray, Z: np.
↪ndarray):
    Y = np.empty((1, 128), dtype="float32")
    for i in range(1):
        for j in range(128):
            for k in range(784):
                if k == 0:
                    Y[i, j] = 0
                Y[i, j] = Y[i, j] + X[i, k] * W[j, k]

    for i in range(1):
        for j in range(128):
            Z[i, j] = Y[i, j] + B[j]


def lnumpy_relu0(X: np.ndarray, Y: np.ndarray):
     for i in range(1):
        for j in range(128):
            Y[i, j] = np.maximum(X[i, j], 0)

def lnumpy_linear1(X: np.ndarray, W: np.ndarray, B: np.ndarray, Z: np.
↪ndarray):
    Y = np.empty((1, 10), dtype="float32")
    for i in range(1):
        for j in range(10):
            for k in range(128):
                if k == 0:
                    Y[i, j] = 0
                Y[i, j] = Y[i, j] + X[i, k] * W[j, k]

    for i in range(1):
        for j in range(10):
            Z[i, j] = Y[i, j] + B[j]


def lnumpy_mlp(data, w0, b0, w1, b1):
    lv0 = np.empty((1, 128), dtype="float32")
    lnumpy_linear0(data, w0, b0, lv0)

    lv1 = np.empty((1, 128), dtype="float32")
    lnumpy_relu0(lv0, lv1)

    out = np.empty((1, 10), dtype="float32")
    lnumpy_linear1(lv1, w1, b1, out)
    return out

result =lnumpy_mlp(
    img.reshape(1, 784),
    mlp_params["w0"],
    mlp_params["b0"],
    mlp_params["w1"],
    mlp_params["b1"])
```

(continues on next page)

(continued from previous page)

```
pred_kind = result.argmax(axis=1)
print("Low-level Numpy MLP Prediction:", class_names[pred_kind[0]])
```

```
Low-level Numpy MLP Prediction: Sneaker
```

3.4 Constructing an End to End IRModule in TVMScript

With the low-level NumPy example in mind, now we are ready to introduce an MLC abstraction for the end-to-end model execution. The code block below shows a TVMScript implementation of the model.

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def relu0(x: T.handle, y: T.handle):
        n = T.int64()
        X = T.match_buffer(x, (1, n), "float32")
        Y = T.match_buffer(y, (1, n), "float32")
        for i, j in T.grid(1, n):
            with T.block("Y"):
                vi, vj = T.axis.remap("SS", [i, j])
                Y[vi, vj] = T.max(X[vi, vj], T.float32(0))

    @T.prim_func
    def linear0(x: T.handle,
                w: T.handle,
                b: T.handle,
                z: T.handle):
        m, n, k = T.int64(), T.int64(), T.int64()
        X = T.match_buffer(x, (1, m), "float32")
        W = T.match_buffer(w, (n, m), "float32")
        B = T.match_buffer(b, (n, ), "float32")
        Z = T.match_buffer(z, (1, n), "float32")
        Y = T.alloc_buffer((1, n), "float32")
        for i, j, k in T.grid(1, n, m):
            with T.block("Y"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + X[vi, vk] * W[vj, vk]
        for i, j in T.grid(1, n):
            with T.block("Z"):
                vi, vj = T.axis.remap("SS", [i, j])
                Z[vi, vj] = Y[vi, vj] + B[vj]

    @R.function
    def main(x: R.Tensor((1, "m"), "float32"),
             w0: R.Tensor(("n", "m"), "float32"),
             b0: R.Tensor(("n", ), "float32"),
             w1: R.Tensor(("k", "n"), "float32"),
             b1: R.Tensor(("k", ), "float32")):
        m, n, k = T.int64(), T.int64(), T.int64()
```

(continues on next page)

(continued from previous page)

```
        with R.dataflow():
            lv0 = R.call_dps_packed("linear0", (x, w0, b0), R.Tensor((1, n),
↪"float32"))
            lv1 = R.call_dps_packed("relu0", (lv0, ), R.Tensor((1, n),
↪"float32"))
            out = R.call_dps_packed("linear0", (lv1, w1, b1), R.Tensor((1, k),
↪ "float32"))
            R.output(out)
        return out
```

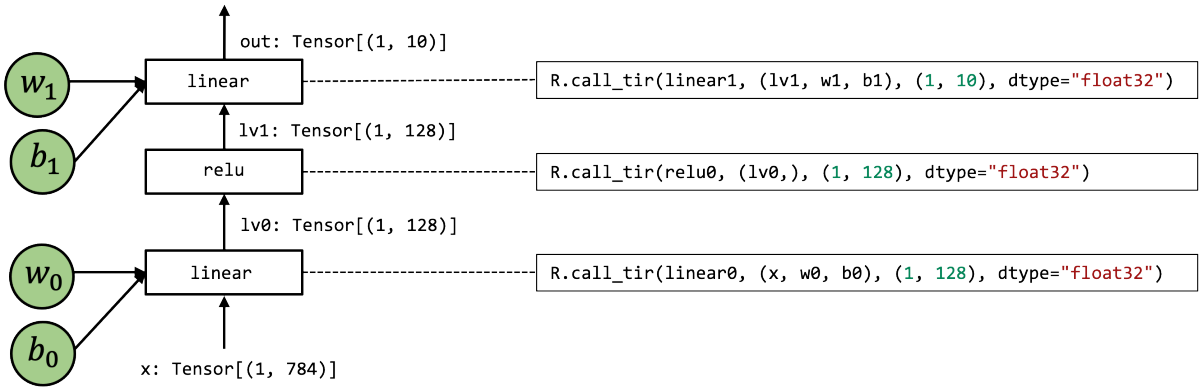
The above code contains kinds of functions: the primitive tensor functions (`T.prim_func`) that we saw in the last lecture and a new `R.function` (relax function). Relax function is a new type of abstraction representing high-level neural network executions.

Again it is helpful to see the TVMScript code and low-level numpy code side-by-side and check the corresponding elements, and we are going to walk through each of them in detail. Since we already learned about primitive tensor functions, we are going to focus on the high-level execution part.

```
@tvm.script.ir_module
class MyModule:
    @R.function
    def main(x: Tensor((1, 784), "float32"),
             w0: Tensor((128, 784), "float32"),
             b0: Tensor((128,), "float32"),
             w1: Tensor((10, 128), "float32"),
             b1: Tensor((10,), "float32")):
        with R.dataflow():
            lv0 = R.call_tir(linear0, (x, w0, b0), (1, 128), dtype="float32")
            lv1 = R.call_tir(relu0, (lv0,), (1, 128), dtype="float32")
            out = R.call_tir(linear1, (lv1, w1, b1), (1, 10), dtype="float32")
            relax.output(out)
        return out
```

```
def lnumpy_mlp(data, w0, b0, w1, b1):
    lv0 = np.empty((1, 128), dtype="float32")
    lnumpy_linear0(data, w0, b0, lv0)

    lv1 = np.empty((1, 128), dtype="float32")
    lnumpy_relu0(lv0, lv1)

    out = np.empty((1, 10), dtype="float32")
    lnumpy_linear1(lv1, w1, b1, out)
    return out
```

3.4.1 Computational Graph View



It is usually helpful to use graph to visualize high-level model executions. The above figure is a graph-view of the `main` function:

- Each of the box in the graph corresponds to computation operations.
- The arrows correspond to the input-output of the intermediate tensors.

We have seen this kind of visualization in earlier lectures. The graph itself can be viewed as a type of abstraction, and it is commonly known as **computational graph** in machine learning frameworks.

3.4.2 `call_dps_packed` Construct

One thing that you may have noticed is that each step of operations in the computational graph contains an `R.call_dps_packed` operation. This is the operation that brings in the tensor primitive functions

```
lv0 = R.call_dps_packed(linear0, (x, w0, b0), (1, 128), dtype="float32")
```

To explain what does `R.call_dps_packed` mean, let us review an equivalent low-level numpy implementation of the operation, as follows:

```
def lnumpy_call_dps_packed(prim_func, inputs, shape, dtype):
    res = np.empty(shape, dtype=dtype)
    prim_func(*inputs, res)
    return res
```

Specifically, call_dps_packed takes in a primitive function (`prim_func`) a list of inputs. Then what it does is allocate an output tensor `res`, then pass the inputs and the output to the `prim_func`. After executing `prim_func` the result is populated in `res`, then we can return the result.

Note that `lnumpy_call_dps_packed` is only a reference implementation to show the meaning of `R.call_dps_packed`. In practice, there can be different low-level ways to optimize the execution. For example, we might choose to allocate all the output memories ahead of time and then run the execution, which we will cover in future lectures.

A natural question that one could ask is why do we need `call_dps_packed` construct? This is because our primitive tensor functions take the following calling convention.

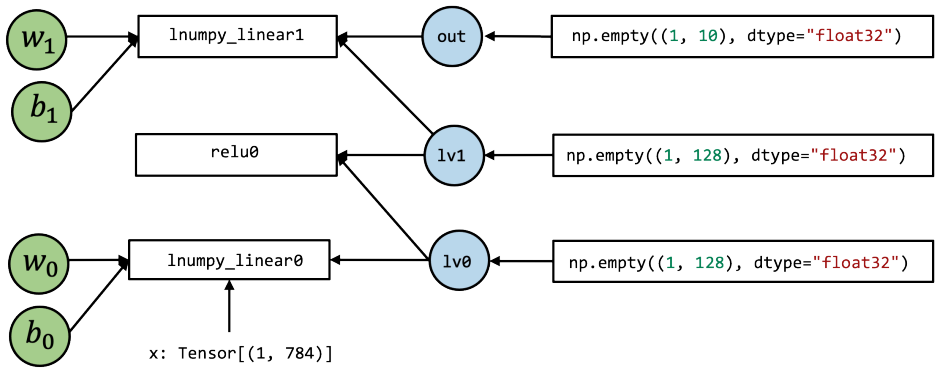
```
def low_level_prim_func(in0, in1, ..., out):
    # implementations
```

This convention is called **destination passing**. The idea is that input and output are explicitly allocated outside and passed to the low-level primitive function. This style is commonly used in low-level library designs, so higher-level frameworks can handle that memory allocation decision. Note that not all tensor operations can be presented in this style (specifically, there are operations whose output shape depends on the input). Nevertheless, in common practice, it is usually helpful to write the low-level function in this style when possible.

While it is possible to assemble the destination passing convention function together by explicitly allocating intermediate results and calling each function, it is hard to turn the following code into computational graph form.

```
def lnumpy_mlp(data, w0, b0, w1, b1):
    lv0 = np.empty((1, 128), dtype="float32")
    lnumpy_linear0(data, w0, b0, lv0)

    lv1 = np.empty((1, 128), dtype="float32")
    lnumpy_relu0(lv0, lv1)

    out = np.empty((1, 10), dtype="float32")
    lnumpy_linear1(lv1, w1, b1, out)
    return out
```

A "failed attempt" to represent low-level NumPy code with explicit allocation in a "computational graph"-like form



We can certainly try a bit :) The above figure is one possible "failed attempt" to fit the `lnumpy_mlp` into a "computational graph-like" form by simply connecting function inputs to the function.

We can find that it lost a few nice properties of the previous computational graphs. Specifically, a computational graph usually has the following properties

- Every input edge to the box corresponds to the input to the operation.
- Every outgoing edge corresponds to the output of the operations.
- Each operation can be reordered arbitrarily up to the topological order of the edges.

Of course, we can still generalize the graph definition by introducing the input edge and output edge, and that can complicate the possible transformations associated with the abstraction.

So coming back to `call_dps_packed`, the key insight here is that we want to hide possible allocation or explicit writing to the functions. In a more formal term, we want the function to be **pure** or **side-effect free**.

A function is **pure** or **side-effect free** if: it only reads from its inputs and returns the result via its output, it will not change other parts of the program (such as incrementing a global counter).

call_dps_packed is a way for us to hide these details of calling into low-level primitive functions and expose them into a computational graph.

We can also see `call_dps_packed` in action in the low-level numpy as well. Now we have defined the `lnumpy_call_dps_packed`, we can rewrite the low-level numpy execution code as:

```
def lnumpy_mlp_with_call_dps_packed(data, w0, b0, w1, b1):
    lv0 = lnumpy_call_dps_packed(lnumpy_linear0, (data, w0, b0), (1, 128),␣
↪dtype="float32")
    lv1 = lnumpy_call_dps_packed(lnumpy_relu0, (lv0, ), (1, 128), dtype=
↪"float32")
    out = lnumpy_call_dps_packed(lnumpy_linear1, (lv1, w1, b1), (1, 10),␣
↪dtype="float32")
    return out

result = lnumpy_mlp_with_call_dps_packed(
    img.reshape(1, 784),
    mlp_params["w0"],
    mlp_params["b0"],
    mlp_params["w1"],
```

(continues on next page)

(continued from previous page)

```
    mlp_params["b1"])

pred_kind = np.argmax(result, axis=1)
print("Low-level Numpy with CallTIR Prediction:", class_names[pred_kind[0]])
```

```
Low-level Numpy with CallTIR Prediction: Sneaker
```

In practice, the lowest-level implementation will have explicit memory allocations, so `call_dps_packed` mainly serves as a purpose for us to continue to do some high-level transformations before we generate the actual implementation.

3.4.3 Dataflow Block

Another important element in a relax function is the `R.dataflow()` scope annotation.

```
with R.dataflow():
    lv0 = R.call_dps_packed("linear0", (x, w0, b0), R.Tensor((1, n), "float32
↪"))
    lv1 = R.call_dps_packed("relu0", (lv0, ), R.Tensor((1, n), "float32"))
    out = R.call_dps_packed("linear0", (lv1, w1, b1), R.Tensor((1, k),
↪"float32"))
    R.output(out)
```

This connects back to the **computational graph** discussion we had in the last section. Recall that ideally, each computational graph operation should be side effect free.

What if we still want to introduce operations that contains side effect? A dataflow block is a way for us to mark the computational graph regions of the program. Specifically, within a dataflow block, all the operations need to be side-effect free. Outside a dataflow block, the operations can contain side-effect. The program below is an example program that contains two dataflow blocks.

```
@R.function
def main(x: R.Tensor((1, "m"), "float32"),
         w0: R.Tensor(("n", "m"), "float32"),
         b0: R.Tensor(("n", ), "float32"),
         w1: R.Tensor(("k", "n"), "float32"),
         b1: R.Tensor(("k", ), "float32")):
    m, n, k = T.int64(), T.int64(), T.int64()

    with R.dataflow():
        lv0 = R.call_dps_packed("linear0", (x, w0, b0), R.Tensor((1, n),
↪"float32"))
        gv0 = R.call_dps_packed("relu0", (lv0, ), R.Tensor((1, n), "float32"))
        R.output(gv0)

    with R.dataflow():
        out = R.call_dps_packed("linear0", (gv0, w1, b1), R.Tensor((1, k),
↪"float32"))
        R.output(out)
    return out
```

Most of our lectures will only deal with computational graphs (dataflow blocks). But it is good to keep the reason behind in mind.

3.4.4 Section Checkpoint

So far, we have gone through one example instance of relax program and covered most of the elements, including:

- Computational graph view
- `call_dps_packed` construct
- Dataflow block.

These elements should get us started in the end to end model execution and compilation. we will also cover new concepts as we encounter them in later chapters.

3.5 Build and Run the Model

In the last section, we discussed the abstraction that enables us to represent end-to-end model execution. This section introduces how to build and run an IRModule. Let us begin by reviewing the IRModule we have.

```
IPython.display.Code(MyModule.script(), language="python")
```

We call `relax.build` to build this function. Relax is still under development, so some of the APIs may change. Our main goal, though, is to get familiar with the overall MLC flow (Construct, transform, build) for end-to-end models.

```
ex = relax.build(MyModule, target="llvm")
type(ex)
```

```
tvm.relax.vm_build.Executable
```

The build function will give us an executable. We can initialize a virtual machine executor that enables us to run the function. Additionally, we will pass in a second argument, indicating which device we want to run the end-to-end executions on.

```
vm = relax.VirtualMachine(ex, tvm.cpu())
```

Now we are ready to run the model. We begin by constructing tvm NDArray that contains input data and weights.

```
data_nd = tvm.nd.array(img.reshape(1, 784))
nd_params = {k: tvm.nd.array(v) for k, v in mlp_params.items()}
```

Then we can run the main function by passing in the input arguments and weights.

```
nd_res = vm["main"](data_nd,
                    nd_params["w0"],
```

(continues on next page)

(continued from previous page)

```
                    nd_params["b0"],
                    nd_params["w1"],
                    nd_params["b1"])
print(nd_res)
```

```
[[-21.10578   -30.078619  -16.665087  -15.054921  -26.568047   -0.322654
  -25.358414   20.298117   -2.9159853 -18.29347  ]]
```

The main function returns the prediction result, and we can then call `nd_res.numpy()` to convert it to numpy array, and take argmax to get the class label.

```
pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModule Prediction:", class_names[pred_kind[0]])
```

```
MyModule Prediction: Sneaker
```

3.6 Integrate Existing Libraries in the Environment

In the last section, we showed how to build an IRModule that contains both the primitive function implementations as well as the high-level computational graph part. In many cases, we may be interested in integrating existing library functions into the MLC process.

The IRModule shows an example on how to do that.

```
@tvm.script.ir_module
class MyModuleWithExternCall:
    @R.function
    def main(x: R.Tensor((1, "m"), "float32"),
             w0: R.Tensor(("n", "m"), "float32"),
             b0: R.Tensor(("n", ), "float32"),
             w1: R.Tensor(("k", "n"), "float32"),
             b1: R.Tensor(("k", ), "float32")):
        # block 0
        m, n, k = T.int64(), T.int64(), T.int64()
        with R.dataflow():
            lv0 = R.call_dps_packed("env.linear", (x, w0, b0), R.Tensor((1,␣
↪n), "float32"))
            lv1 = R.call_dps_packed("env.relu", (lv0, ), R.Tensor((1, n),
↪"float32"))
            out = R.call_dps_packed("env.linear", (lv1, w1, b1), R.Tensor((1,␣
↪k), "float32"))
            R.output(out)
        return out
```

Note that we now directly pass in strings in `call_dps_packed`

```
R.call_dps_packed("env.linear", (x, w0, b0), R.Tensor((1, n), "float32"))
```

These strings are names of runtime functions that we expect to exist during model execution.

3.6.1 Registering Runtime Function

In order to be able to execute the code that calls into external functions, we need to register the corresponding functions. The code block below registers two implementations of the functions.

```
@tvm.register_func("env.linear", override=True)
def torch_linear(x: tvm.nd.NDArray,
                 w: tvm.nd.NDArray,
                 b: tvm.nd.NDArray,
                 out: tvm.nd.NDArray):
    x_torch = torch.from_dlpack(x)
    w_torch = torch.from_dlpack(w)
    b_torch = torch.from_dlpack(b)
    out_torch = torch.from_dlpack(out)
    torch.mm(x_torch, w_torch.T, out=out_torch)
    torch.add(out_torch, b_torch, out=out_torch)

@tvm.register_func("env.relu", override=True)
def lnumpy_relu(x: tvm.nd.NDArray,
                out: tvm.nd.NDArray):
    x_torch = torch.from_dlpack(x)
    out_torch = torch.from_dlpack(out)
    torch.maximum(x_torch, torch.Tensor([0.0]), out=out_torch)
```

In the above code, we use the `from_dlpack` to convert a TVM NDArray to a torch NDArray. Note that this is a zero-copy conversion, which means the torch array shares the underlying memory with the TVM NDArray. DLPack is a common exchange standard that allows different frameworks to exchange Tensor/NDArray without being involved in data copy. The `from_dlpack` API is supported by multiple frameworks and is part of the python array API standard. If you are interested, you can read more here[1].

In this particular function, we simply piggyback PyTorch's implementation. In real-world settings, we can use a similar mechanism to redirect calls onto specific libraries, such as cuDNN or our own library implementations.

This particular example performs the registration in python. In reality, we can register functions in different languages (such as C++) that do not have a python dependency. We will cover more in future lectures.

3.6.2 Build and Run

Now we can build and run `MyModuleWithExternCall`, and we can verify that we get the same result.

```
ex = relax.build(MyModuleWithExternCall, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())

nd_res = vm["main"](data_nd,
                    nd_params["w0"],
                    nd_params["b0"],
                    nd_params["w1"],
                    nd_params["b1"])

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModuleWithExternCall Prediction:", class_names[pred_kind[0]])
```

[1] https://dmlc.github.io/dlpack/latest/python_spec.html

```
MyModuleWithExternCall Prediction: Sneaker
```

3.7 Mixing TensorIR Code and Libraries

In the last example, we build an IRModule where all primitive operations are dispatched to library functions. Sometimes it can be helpful to have a mixture of both.

```
@tvm.script.ir_module
class MyModuleMixture:
    @T.prim_func
    def linear0(x: T.handle,
                w: T.handle,
                b: T.handle,
                z: T.handle):
        m, n, k = T.int64(), T.int64(), T.int64()
        X = T.match_buffer(x, (1, m), "float32")
        W = T.match_buffer(w, (n, m), "float32")
        B = T.match_buffer(b, (n, ), "float32")
        Z = T.match_buffer(z, (1, n), "float32")
        Y = T.alloc_buffer((1, n), "float32")
        for i, j, k in T.grid(1, n, m):
            with T.block("Y"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + X[vi, vk] * W[vj, vk]
        for i, j in T.grid(1, n):
            with T.block("Z"):
                vi, vj = T.axis.remap("SS", [i, j])
                Z[vi, vj] = Y[vi, vj] + B[vj]

    @R.function
    def main(x: R.Tensor((1, "m"), "float32"),
             w0: R.Tensor(("n", "m"), "float32"),
             b0: R.Tensor(("n", ), "float32"),
             w1: R.Tensor(("k", "n"), "float32"),
             b1: R.Tensor(("k", ), "float32")):
        m, n, k = T.int64(), T.int64(), T.int64()
        with R.dataflow():
            lv0 = R.call_dps_packed("linear0", (x, w0, b0), R.Tensor((1, n),
↪"float32"))
            lv1 = R.call_dps_packed("env.relu", (lv0, ), R.Tensor((1, n),
↪"float32"))
            out = R.call_dps_packed("env.linear", (lv1, w1, b1), R.Tensor((1,␣
↪k), "float32"))
            R.output(out)
        return out
```

The above code block shows an example where linear0 is still implemented in TensorIR, while the rest of the functions are redirected to library functions. We can build and run to validate the result.

```
ex = relax.build(MyModuleMixture, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())

nd_res = vm["main"](data_nd,
                    nd_params["w0"],
                    nd_params["b0"],
                    nd_params["w1"],
                    nd_params["b1"])

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModuleMixture Prediction:", class_names[pred_kind[0]])
```

```
MyModuleMixture Prediction: Sneaker
```

3.8 Bind Parameters to IRModule

In all the examples so far, we construct the main function by passing in the parameters explicitly. In many cases, it is usually helpful to bind the parameters as constants attached to the IRModule. The following code created the binding by matching the parameter names to the keys in nd_params.

```
MyModuleWithParams = relax.transform.BindParams("main", nd_
↪params)(MyModuleMixture)
IPython.display.Code(MyModuleWithParams.script(), language="python")
```

In the above script, `meta[relay.Constant][0]` corresponds to an implicit dictionary that stores the constant (which is not shown as part of the script but still is part of the IRModule). If we build the transformed IRModule, we can now invoke the function by just passing in the input data.

```
ex = relax.build(MyModuleWithParams, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())

nd_res = vm["main"](data_nd)

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModuleWithParams Prediction:", class_names[pred_kind[0]])
```

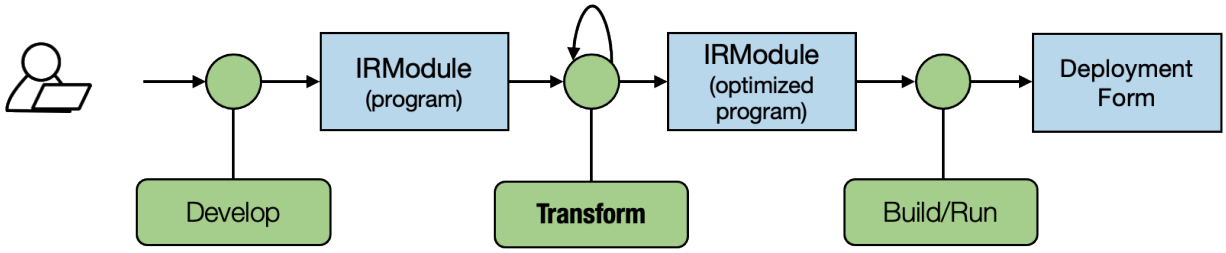
```
MyModuleWithParams Prediction: Sneaker
```

3.9 Discussions

In this chapter, we have discussed many ways to describe the end-to-end model execution. One thing we may have noticed is that we are coming back to the theme of **abstraction and implementation**

- Both the TensorIR function and library functions follow the same destination passing style. As a result, we can simply replace invocation from one to another in our examples.
- We may use different ways to represent the computation at different stages of the MLC process.

So far, we have touched on a few ways to transform the end-to-end IRModule (e.g. parameter binding). Let us come back to the following common theme of MLC: MLC process is about representing the execution in possibly different abstractions and transforming among them.



There are many possible transformations in the end-to-end execution. For example, we can take the TensorIR function in MyModuleMixture and change the `linear0` function using the schedule operations taught in the last lecture. In other instances, we might want to transform high-level model executions into a mixture of library function calls and TensorIR functions.

As an exercise, spend some time thinking about what kinds of transformations you might want to perform on an IRModule. We will also cover more transformations in the future.

In this chapter, we construct the IRModule by hand. In practice, a real neural network model can contain hundreds of layers, so it is infeasible to write things out manually. Still, the script format is helpful for us to peek into what is going on and do interactive developments. We will also learn about more ways to construct IRModule in future episodes programmatically.

3.10 Summary

- Computational graph abstraction helps to stitch primitive tensor functions together for end-to-end execution.
- Key elements of relax abstraction include
  - call_dps_packed construct that embeds destination passing style primitive function into the computational graph
  - dataflow block
- Computational graph allows call into both environment library functions and TensorIR functions.

4 | Automatic Program Optimization

4.1 Prelude

In the past chapters, we learned about how to build primitive tensor functions and connect them to form end-to-end model executions. There are three primary types of abstractions we have used so far.

- A computational graph view that drives the high-level executions.
- Abstraction for primitive tensor functions.
- Library function calls via environment function registration.

All of these elements are encapsulated in an IRModule. Most of the MLC processes can be viewed as transformations among tensor functions.

There are many different ways to transform the same program. This chapter will discuss ways to automate some of the processes.

4.2 Preparations

To begin with, we will import necessary dependencies and create helper functions.

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

```
import IPython


def code2html(code):
    """Helper function to use pygments to turn the code string into␣
↪highlighted html."""
    import pygments
    from pygments.formatters import HtmlFormatter
    from pygments.lexers import Python3Lexer
    formatter = HtmlFormatter()
    html = pygments.highlight(code, Python3Lexer(), formatter)
    return "<style>%s</style>%s\n" % (formatter.get_style_defs(".highlight"),␣
↪html)
```

4.3 Recap: Transform a Primitive Tensor Function.

Let us begin by reviewing what we did in our previous chapters – transforming a single primitive tensor function.

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def main(
        A: T.Buffer((128, 128), "float32"),
        B: T.Buffer((128, 128), "float32"),
        C: T.Buffer((128, 128), "float32"),
    ):
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i, j, k in T.grid(128, 128, 128):
            with T.block("C"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    C[vi, vj] = 0.0
                C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]
```

First, let us define a set of inputs and outputs for evaluation.

```
dtype = "float32"
a_np = np.random.rand(128, 128).astype(dtype)
b_np = np.random.rand(128, 128).astype(dtype)
c_mm = a_np @ b_np
```

We can build and run `MyModule` as follows.

```
a_nd = tvm.nd.array(a_np)
b_nd = tvm.nd.array(b_np)
c_nd = tvm.nd.empty((128, 128), dtype="float32")

lib = tvm.build(MyModule, target="llvm")
f_timer_before = lib.time_evaluator("main", tvm.cpu())
print("Time cost of MyModule: %.3f ms" % (f_timer_before(a_nd, b_nd, c_nd).
↪mean * 1000))
```

```
Time cost of MyModule: 1.595 ms
```

Next, we transform `MyModule` a bit by reorganizing the loop access pattern.

```
def schedule_mm(sch: tvm.tir.Schedule, jfactor=4):
    block_C = sch.get_block("C", "main")
    i, j, k = sch.get_loops(block=block_C)
    j_0, j_1 = sch.split(loop=j, factors=[None, jfactor])
    sch.reorder(i, j_0, k, j_1)
    sch.decompose_reduction(block_C, k)
    return sch
```

```
sch = tvm.tir.Schedule(MyModule)
sch = schedule_mm(sch)
IPython.display.HTML(code2html(sch.mod.script()))
```

Then we can build and run the re-organized program.

```
lib = tvm.build(sch.mod, target="llvm")
f_timer_after = lib.time_evaluator("main", tvm.cpu())
print("Time cost of MyModule=>schedule_mm: %.3f ms" % (f_timer_after(a_nd, b_
↪nd, c_nd).mean * 1000))
```

```
Time cost of MyModule=>schedule_mm: 0.238 ms
```

4.3.1 Transformation Trace

Besides `sch.mod` field, another thing `tir.Schedule` offers is a trace field that can be used to show the steps involved to get to the transformed module. We can print it out using the following code.

```
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  l4, l5 = sch.split(loop=l2, factors=[None, 4], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l4, l3, l5)
  b6 = sch.decompose_reduction(block=b0, loop=l3)
```

```
def schedule_mm(sch: tvm.tir.Schedule, jfactor=4):
    block_C = sch.get_block("C", "main")
    i, j, k = sch.get_loops(block=block_C)
    j_0, j_1 = sch.split(loop=j, factors=[None, jfactor])
    sch.reorder(i, j_0, k, j_1)
    sch.decompose_reduction(block_C, k)
    return sch
```

The above trace aligns with the transformations we specified in `schedule_mm`. One thing to note is that the trace (plus the original program) gives us a way to completely re-derive the final output program. Let us keep that in mind; we will use trace throughout this chapter as another way to inspect the transformations.

4.4 Stochastic Schedule Transformation

Up until now, we have specified every detail about what transformations we want to make on the original TensorIR program. Many of those choices are based on our understanding of the underlying environment, such as cache and hardware unit.

However, in practice, we may not be able to decide every detail accurately. Instead of doing so, we would like to specify **what are possible ways to transform the program, while leaving out some details**.

One natural way to achieve the goal is to add some stochastic (randomness) elements to our transformations. The following code does that.

```
def stochastic_schedule_mm(sch: tvm.tir.Schedule):
    block_C = sch.get_block("C", "main")
    i, j, k = sch.get_loops(block=block_C)
    j_factors = sch.sample_perfect_tile(loop=j, n=2)
    j_0, j_1 = sch.split(loop=j, factors=j_factors)
    sch.reorder(i, j_0, k, j_1)
    sch.decompose_reduction(block_C, k)
    return sch
```

```
def stochastic_schedule_mm(sch: tir.Schedule):
    block_C = sch.get_block("C", "main")
    i, j, k = sch.get_loops(block=block_C)
    j_factors = sch.sample_perfect_tile(loop=j, n=2)
    j_0, j_1 = sch.split(loop=j, factors=j_factors)
    sch.reorder(i, j_0, k, j_1)
    sch.decompose_reduction(block_C, k)
    return sch
```

```
def schedule_mm(sch: tir.Schedule, jfactor=4):
    block_C = sch.get_block("C", "main")
    i, j, k = sch.get_loops(block=block_C)
    j_0, j_1 = sch.split(loop=j, factors=[None, jfactor])
    sch.reorder(i, j_0, k, j_1)
    sch.decompose_reduction(block_C, k)
    return sch
```

Let us compare `stochastic_schedule_mm` and `schedule_mm` side by side. We can find that the only difference is how to specify `j_factors`. In the case of `schedule_mm`, `j_factors` is passed in as a parameter specified by us. In the case of `stochastic_schedule_mm`, it comes from `sch.sample_perfect_tile`.

As the name suggests, `sch.sample_perfect_tile` tries to draw random numbers to fill in `j_factors`. It samples factors such that they perfectly split the loop. For example, when the original loop size is `128`, possible ways to split the loop include: `[8, 16]`, `[32, 4]`, `[2, 64]` (note `8 * 16 = 32 * 4 = 2 * 64 = 128`).

Let us first try to see what is the effect of `stochastic_schedule_mm` by running the following code-block. Try to run the following code block multiple times and observe the outcome difference. You might find that the loop bound of `j_1` changes each time we run the code-block.

```
sch = tvm.tir.Schedule(MyModule)
sch = stochastic_schedule_mm(sch)

IPython.display.HTML(code2html(sch.mod.script()))
```

What is happening here is that each time we run `stochastic_schedule_mm` it draws a different `j_factors` randomly. We can print out the trace of the latest one to see the decisions we made in sampling.

```
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[16, 8])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
```

(continues on next page)

(continued from previous page)

```
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
```

When we look at the trace, pay close attention to the `decision=[...]` part of `sample_perfect_tile`. They correspond to the value that the `sampling_perfect_tile` picked in our last call to `stochastic_schedule_mm`.

As an alternative way to look at different samples of `stochastic_schedule_mm`, we can run the following block multiple times and look at the trace.

```
sch = tvm.tir.Schedule(MyModule)
sch = stochastic_schedule_mm(sch)
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[8, 16])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
```

4.4.1 Deep Dive into Stochastic Transformation

Now let us take a deeper dive into what happened in stochastic schedule transformations. We can find that it is a simple generalization of the original deterministic transformations, with two additional elements:

- Random variables that come from `sample_perfect_tile` and other sampling operations that we did not cover in the example.
- Schedule operations that take action depending on the random variables.

Let us try to run the stochastic transformation step by step.

```
sch = tvm.tir.Schedule(MyModule)
block_C = sch.get_block("C", "main")
i, j, k = sch.get_loops(block=block_C)
j_factors = sch.sample_perfect_tile(loop=j, n=2)
```

```
type(j_factors[0])
```

```
tvm.tir.expr.Var
```

Elements in the `j_factors` are not real integer numbers. Instead, they are **symbolic variables** that refer to a random variable being sampled. We can pass these variables to the transformation API to specify choices such as factor values.

```
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[32, 4])
```

The schedule trace keeps track of the choices of these symbolic variables in the `decisions` field. So follow-up steps will be able to look up these choices to decide how to split the loop.

```
IPython.display.HTML(code2html(sch.mod.script()))
```

If we look at the code at the current time point, we can find that the module remains the same since we only sampled the random variables but have not yet made any transformation actions based on them.

Let us now take some of the actions:

```
j_0, j_1 = sch.split(loop=j, factors=j_factors)
sch.reorder(i, j_0, k, j_1)
```

These actions are recorded in the following trace.

```
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[32, 4])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
```

If we retake a look at the code, the transformed module now corresponds to the updated versions after the actions are taken.

```
IPython.display.HTML(code2html(sch.mod.script()))
```

We can do some further transformations to get to the final state.

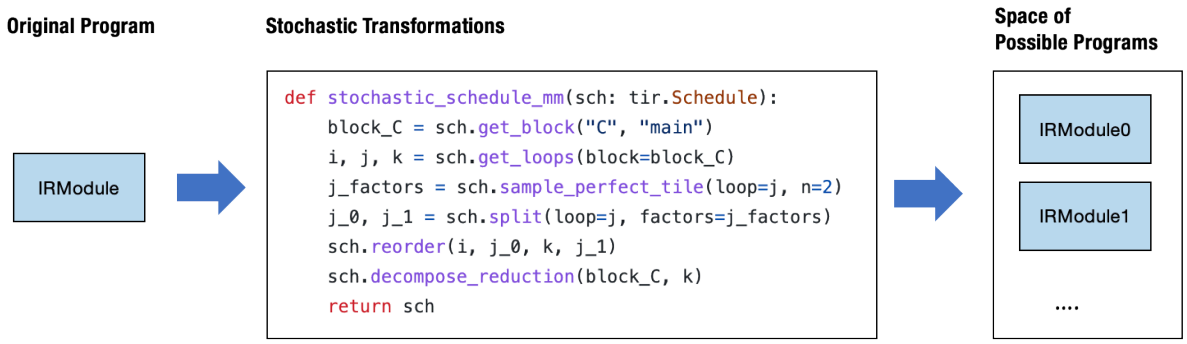
```
sch.reorder(i, j_0, k, j_1)
sch.decompose_reduction(block_C, k)
```

```
tir.BlockRV(0x10a93760)
```

```
IPython.display.HTML(code2html(sch.mod.script()))
```

4.5 Search Over Stochastic Transformations

One thing that you might realize is that `stochastic_schedule_mm` create a **search space of possible programs** depending on the specific decisions made at each sampling step.



Coming back to our initial intuition, we want to be able to specify a set of **possible programs** instead of one program. `stochastic_schedule_mm` did exactly that. Of course, one natural question to ask next is what is the best choice.

We will need a search algorithm to do that. To show what can be done here, let us first try the most straightforward search algorithm – random search, in the following code block. It tries to run `stochastic_schedule_mm` repetitively, gets a transformed module, runs benchmark, then book keep the best one in history.

```
def random_search(mod: tvm.IRModule, num_trials=5):
    best_result = None
    best_sch = None

    for i in range(num_trials):
        sch = stochastic_schedule_mm(tvm.tir.Schedule(mod))
        lib = tvm.build(sch.mod, target="llvm")
        f_timer_after = lib.time_evaluator("main", tvm.cpu())
        result = f_timer_after(a_nd, b_nd, c_nd).mean

        print("=====Attempt %d, time-cost: %.3f ms====" % (i, result * 1000))
        print(sch.trace)

        # book keep the best result so far
        if best_result is None or result < best_result:
            best_result = result
            best_sch = sch

    return best_sch

sch = random_search(MyModule)
```

```
=====Attempt 0, time-cost: 0.161 ms====
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
```

(continues on next page)

(continued from previous page)

```
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[16, 8])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
=====Attempt 1, time-cost: 0.483 ms====
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[64, 2])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
=====Attempt 2, time-cost: 0.162 ms====
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[16, 8])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
=====Attempt 3, time-cost: 0.127 ms====
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[8, 16])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
=====Attempt 4, time-cost: 0.238 ms====
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[32, 4])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
```

If we run the code, we can find that it goes over a few choices and then returns the best run throughout five trials.

```
print(sch.trace)
```

```
# from tvm import tir
def apply_trace(sch: tir.Schedule) -> None:
  b0 = sch.get_block(name="C", func_name="main")
  l1, l2, l3 = sch.get_loops(block=b0)
  v4, v5 = sch.sample_perfect_tile(loop=l2, n=2, max_innermost_factor=16,␣
↪decision=[8, 16])
  l6, l7 = sch.split(loop=l2, factors=[v4, v5], preserve_unit_iters=True,␣
↪disable_predication=False)
  sch.reorder(l1, l6, l3, l7)
  b8 = sch.decompose_reduction(block=b0, loop=l3)
```

In practice, we use smarter algorithms. We also need to provide additional utilities, such as benchmarking on remote devices, if we are interested in optimization for other devices. TVM's meta schedule API provides these additional capabilities.

`meta_schedule` is the namespace that comes to support search over a space of possible transformations. There are many additional things that meta-schedule do behind the scene:

- Parallel benchmarking across many processes.
- Use cost models to avoid benchmarking each time.
- Evolutionary search on the traces instead of randomly sampling at each time.

Despite these magics, the key idea remains the same: **use stochastic transformation to specify a search space of good programs, ``tune_tir`` API helps to search and find an optimized solution within the search space**.

```
from tvm import meta_schedule as ms

database = ms.tune_tir(
    mod=MyModule,
    target="llvm --num-cores=1",
    max_trials_global=64,
    num_trials_per_iter=64,
    space=ms.space_generator.ScheduleFn(stochastic_schedule_mm),
    work_dir="./tune_tmp",
)

sch = ms.tir_integration.compile_tir(database, MyModule, "llvm --num-cores=1")
```

```
2025-07-10 12:32:04 [INFO] Logging directory: ./tune_tmp/logs
2025-07-10 12:32:10 [INFO] LocalBuilder: max_workers = 10
2025-07-10 12:32:11 [INFO] LocalRunner: max_workers = 1
2025-07-10 12:32:12 [INFO] [task_scheduler.cc:159] Initializing Task #0: "main
↪"
```

```
2025-07-10 12:32:12 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
↪------------------------
  0 | main | 4194304 |      1 |            N/A |          N/A |              ␣
↪    N/A |      0 |
-------------------------------------------------------------------------------
↪------------------------
Total trials: 0
Total latency (us): 0


Total trials: 0
Total latency (us): 0

2025-07-10 12:32:12 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:12 [INFO] [task_scheduler.cc:193] Sending 5 sample(s) to␣
↪builder
2025-07-10 12:32:14 [INFO] [task_scheduler.cc:195] Sending 5 sample(s) to␣
↪runner
2025-07-10 12:32:15 [DEBUG] XGB iter   0: tr-p-rmse: 0.292503       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.313647       tr-rmse: 0.313647
2025-07-10 12:32:15 [DEBUG] XGB iter  25: tr-p-rmse: 0.096080       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.056833       tr-rmse: 0.056833
2025-07-10 12:32:15 [DEBUG] XGB iter  50: tr-p-rmse: 0.095582       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.056823       tr-rmse: 0.056823
2025-07-10 12:32:16 [DEBUG] XGB iter  75: tr-p-rmse: 0.095573       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.056823       tr-rmse: 0.056823
2025-07-10 12:32:16 [DEBUG] XGB iter 100: tr-p-rmse: 0.095573       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.056823       tr-rmse: 0.056823
2025-07-10 12:32:16 [DEBUG] XGB stopped. Best iteration: [69] tr-p-rmse:0.
↪09557     tr-a-peak@32:1.00000    tr-rmse:0.05682 tr-rmse:0.05682
2025-07-10 12:32:16 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:16 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |             ␣
↪229.4258 |      5 |
-------------------------------------------------------------------------------
↪------------------------
Total trials: 5
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426

2025-07-10 12:32:16 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:193] Sending 0 sample(s) to␣
↪builder
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:195] Sending 0 sample(s) to␣
↪runner
```

(continues on next page)

(continued from previous page)

```
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:17 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪-------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |             ␣
↪229.4258 |      5 |
-------------------------------------------------------------------------------
↪-------------------------
Total trials: 5
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426

2025-07-10 12:32:17 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:193] Sending 0 sample(s) to␣
↪builder
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:195] Sending 0 sample(s) to␣
↪runner
2025-07-10 12:32:17 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:17 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪-------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |             ␣
↪229.4258 |      5 |
-------------------------------------------------------------------------------
↪-------------------------
Total trials: 5
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426

2025-07-10 12:32:17 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:193] Sending 0 sample(s) to␣
↪builder
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:195] Sending 0 sample(s) to␣
↪runner
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:18 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
```

(continues on next page)

(continued from previous page)

```
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |              ␣
↪229.4258 |      5 |
-------------------------------------------------------------------------------
↪------------------------
Total trials: 5
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426

2025-07-10 12:32:18 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:193] Sending 0 sample(s) to␣
↪builder
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:195] Sending 0 sample(s) to␣
↪runner
2025-07-10 12:32:18 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:18 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |              ␣
↪229.4258 |      5 |
-------------------------------------------------------------------------------
↪------------------------
Total trials: 5
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426

2025-07-10 12:32:18 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:19 [INFO] [task_scheduler.cc:260] Task #0 has finished.␣
↪Remaining task(s): 0
```

```
2025-07-10 12:32:19 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪------------------------
  0 | main | 4194304 |      1 |        18.2817 |     229.4258 |              ␣
↪229.4258 |      5 |    Y
-------------------------------------------------------------------------------
↪------------------------
Total trials: 5
```

(continues on next page)

(continued from previous page)

```
Total latency (us): 229.426


Total trials: 5
Total latency (us): 229.426
```

`tune_tir` functions return an optimized schedule found during the tuning process.

```
sch.trace.show()
```

```
IPython.display.HTML(code2html(sch.mod.script()))
```

```
lib = tvm.build(sch.mod, target="llvm")
f_timer_after = lib.time_evaluator("main", tvm.cpu())
print("Time cost of MyModule after tuning: %.3f ms" % (f_timer_after(a_nd, b_
↪nd, c_nd).mean * 1000))
```

```
Time cost of MyModule after tuning: 0.158 ms
```

4.5.1 Leverage Default AutoScheduling

In the last section, we showed how to tune a workload with stochastic transformations that we crafted. Metaschedule comes with its own built-in set of generic stochastic transformations that works for a broad set of TensorIR computations. This approach is also called auto-scheduling, as the search space is generated by the system. We can run it by removing the line `space=ms.space_generator.ScheduleFn(stochastic_schedule_mm)`.

Under the hood, the meta-scheduler analyzes each block's data access and loop patterns and proposes stochastic transformations to the program. We won't go into these generic transformations in this chapter but want to note that they are also just stochastic transformations coupled with an analysis of the code. We can use the same mechanisms learned in the last section to enhance auto-scheduling. We will touch base on this topic in future chapters.

```
database = ms.tune_tir(
    mod=MyModule,
    target="llvm --num-cores=1",
    max_trials_global=64,
    num_trials_per_iter=64,
    work_dir="./tune_tmp",
)
sch = ms.tir_integration.compile_tir(database, MyModule, "llvm --num-cores=1")
```

```
2025-07-10 12:32:19 [INFO] Logging directory: ./tune_tmp/logs
2025-07-10 12:32:19 [INFO] LocalBuilder: max_workers = 10
2025-07-10 12:32:19 [INFO] LocalRunner: max_workers = 1
2025-07-10 12:32:20 [INFO] [task_scheduler.cc:159] Initializing Task #0: "main
↪"
```

```
2025-07-10 12:32:20 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪-------------------------
  0 | main | 4194304 |      1 |            N/A |          N/A |              ␣
↪    N/A |      0 |
-------------------------------------------------------------------------------
↪-------------------------
Total trials: 0
Total latency (us): 0


Total trials: 0
Total latency (us): 0

2025-07-10 12:32:20 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:32:22 [INFO] [task_scheduler.cc:193] Sending 64 sample(s) to␣
↪builder
2025-07-10 12:32:27 [INFO] [task_scheduler.cc:195] Sending 64 sample(s) to␣
↪runner
2025-07-10 12:32:44 [DEBUG] XGB iter   0: tr-p-rmse: 0.356943       tr-a-
↪peak@32: 0.998523  tr-rmse: 0.282413       tr-rmse: 0.282413
2025-07-10 12:32:44 [DEBUG] XGB iter  25: tr-p-rmse: 0.028652       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.309059       tr-rmse: 0.309059
2025-07-10 12:32:45 [DEBUG] XGB iter  50: tr-p-rmse: 0.028655       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.309052       tr-rmse: 0.309052
2025-07-10 12:32:45 [DEBUG] XGB stopped. Best iteration: [24] tr-p-rmse:0.
↪02865     tr-a-peak@32:1.00000    tr-rmse:0.30906 tr-rmse:0.30906
2025-07-10 12:32:45 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:32:45 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪-------------------------
  0 | main | 4194304 |      1 |        65.0726 |      64.4558 |              ␣
↪64.4558 |     64 |
-------------------------------------------------------------------------------
↪-------------------------
Total trials: 64
Total latency (us): 64.4558


Total trials: 64
Total latency (us): 64.4558

2025-07-10 12:32:45 [INFO] [task_scheduler.cc:260] Task #0 has finished.␣
↪Remaining task(s): 0
```

```
2025-07-10 12:32:45 [DEBUG] [task_scheduler.cc:318]
 ID | Name |    FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
↪-------------------------
  0 | main | 4194304 |      1 |         65.0726 |      64.4558 |              ␣
↪64.4558 |     64 |     Y
--------------------------------------------------------------------------------
↪-------------------------
Total trials: 64
Total latency (us): 64.4558


Total trials: 64
Total latency (us): 64.4558
```

```
lib = tvm.build(sch.mod, target="llvm")
f_timer_after = lib.time_evaluator("main", tvm.cpu())
print("Time cost of MyModule after tuning: %.3f ms" % (f_timer_after(a_nd, b_
↪nd, c_nd).mean * 1000))
```

```
Time cost of MyModule after tuning: 0.060 ms
```

The result gets much faster than our original code. We can take a glimpse at the trace and the final code. For the purpose of this chapter, you do not need to understand all the transformations. At a high level, the trace involves:

- More levels of loop tiling transformations.
- Vectorization of intermediate computations.
- Parallelization and unrolling of loops.

```
sch.trace.show()
```

```
IPython.display.HTML(code2html(sch.mod.script()))
```

4.5.2 Section Checkpoint

Let us have a checkpoint about what we have learned so far.

- Stochastic schedule allow us to express "what are the possible transformations".
- Metaschedule's `tune_tir` API helps to find a good solution within the space.
- Metaschedule comes with a default built-in set of stochastic transformations that covers a broad range of search space.

4.6 Putting Things Back to End to End Model Execution

Up until now, we have learned to automate program optimization of a single tensor primitive function. How can we put it back and improve our end-to-end model execution?

From the MLC perspective, the automated search is a modular step, and we just need to replace the original primitive function implementation with the new one provided by the tuned result.

We will reuse the two-layer MLP example from the last chapter.

```
import torch
import torchvision

test_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=1,␣
↪shuffle=True)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

img, label = next(iter(test_loader))
img = img.reshape(1, 28, 28).numpy()
```

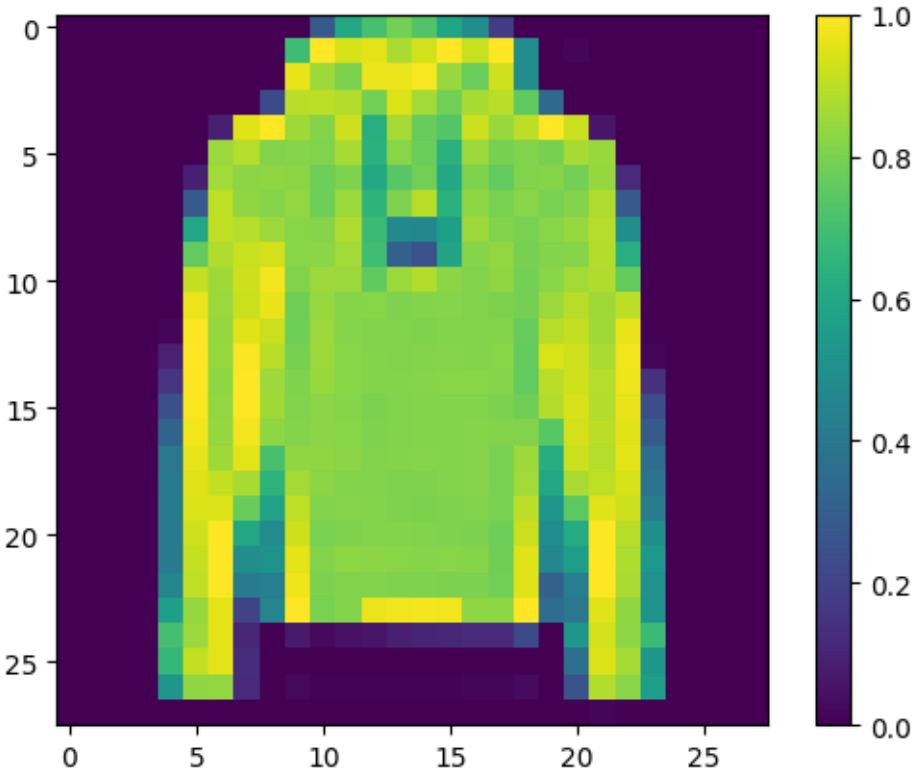
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
```

(continues on next page)

(continued from previous page)

```
100.0%Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw
```

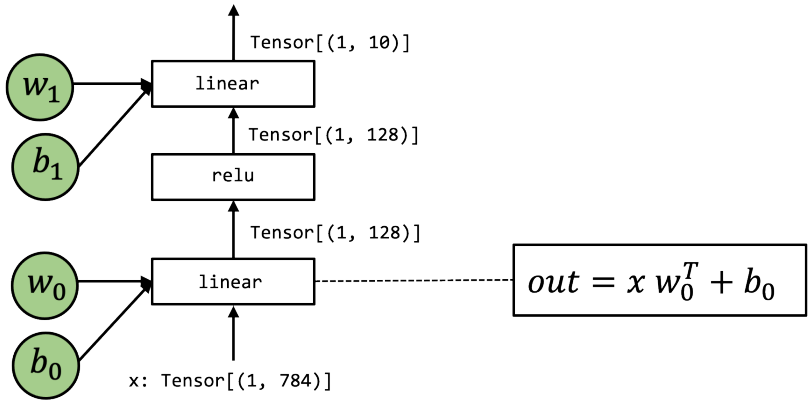
```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(img[0])
plt.colorbar()
plt.grid(False)
plt.show()

print("Class:", class_names[label[0]])
```




```
Class: Pullover
```

We also download pre-packed model parameters that we will use in our examples.

```
# Hide outputs
!wget -nc https://github.com/mlc-ai/web-data/raw/main/models/fasionmnist_mlp_
↪params.pkl
```



As a reminder, the above figure shows the model of interest.

```
import pickle as pkl

mlp_params = pkl.load(open("fasionmnist_mlp_params.pkl", "rb"))

data_nd = tvm.nd.array(img.reshape(1, 784))
nd_params = {k: tvm.nd.array(v) for k, v in mlp_params.items()}
```

Let us use a mixture module where most of the components call into environment function and also come with one TensorIR function `linear0`.

```
@tvm.script.ir_module
class MyModuleMixture:
    @T.prim_func
    def linear0(X: T.Buffer((1, 784), "float32"),
                W: T.Buffer((128, 784), "float32"),
                B: T.Buffer((128,), "float32"),
                Z: T.Buffer((1, 128), "float32")):
        T.func_attr({"global_symbol": "linear0", "tir.noalias": True})
        Y = T.alloc_buffer((1, 128), "float32")
        for i, j, k in T.grid(1, 128, 784):
            with T.block("Y"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = T.float32(0)
                Y[vi, vj] = Y[vi, vj] + X[vi, vk] * W[vj, vk]

        for i, j in T.grid(1, 128):
            with T.block("Z"):
                vi, vj = T.axis.remap("SS", [i, j])
                Z[vi, vj] =  Y[vi, vj] + B[vj]

    @R.function
    def main(x: R.Tensor((1, 784), "float32"),
             w0: R.Tensor((128, 784), "float32"),
             b0: R.Tensor((128,), "float32"),
             w1: R.Tensor((10, 128), "float32"),
             b1: R.Tensor((10,), "float32")):
        with R.dataflow():
            lv0 = R.call_dps_packed("linear0", (x, w0, b0), R.Tensor((1, 128),
```

(continues on next page)

(continued from previous page)

```
↪ dtype="float32"))
            lv1 = R.call_dps_packed("env.relu", (lv0,), R.Tensor((1, 128),␣
↪dtype="float32"))
            out = R.call_dps_packed("env.linear", (lv1, w1, b1), R.Tensor((1,␣
↪10), dtype="float32"))
            R.output(out)
        return out
```

```
@tvm.register_func("env.linear", override=True)
def torch_linear(x: tvm.nd.NDArray,
                 w: tvm.nd.NDArray,
                 b: tvm.nd.NDArray,
                 out: tvm.nd.NDArray):
    x_torch = torch.from_dlpack(x)
    w_torch = torch.from_dlpack(w)
    b_torch = torch.from_dlpack(b)
    out_torch = torch.from_dlpack(out)
    torch.mm(x_torch, w_torch.T, out=out_torch)
    torch.add(out_torch, b_torch, out=out_torch)

@tvm.register_func("env.relu", override=True)
def lnumpy_relu(x: tvm.nd.NDArray,
                out: tvm.nd.NDArray):
    x_torch = torch.from_dlpack(x)
    out_torch = torch.from_dlpack(out)
    torch.maximum(x_torch, torch.Tensor([0.0]), out=out_torch)
```

We can bind the parameters and see if it gives the correct prediction.

```
MyModuleWithParams = relax.transform.BindParams("main", nd_
↪params)(MyModuleMixture)
```

```
ex = relax.build(MyModuleWithParams, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())

nd_res = vm["main"](data_nd)

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModuleWithParams Prediction:", class_names[pred_kind[0]])
```

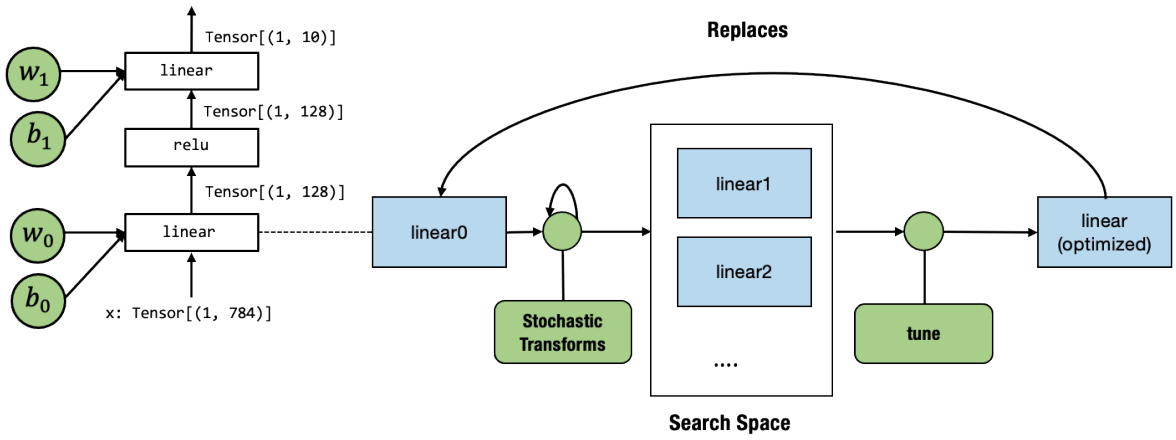
```
MyModuleWithParams Prediction: Pullover
```

The following code evaluates the run time cost of the module before the transformation. Note that because this is a small model, the number can fluctuate a bit between runs, so we just need to read the overall magnitude.

```
ftimer = vm.module.time_evaluator("main", tvm.cpu(), number=100)

print("MyModuleWithParams time-cost: %g ms" % (ftimer(data_nd).mean * 1000))
```

```
MyModuleWithParams time-cost: 0.0664007 ms
```

We are now ready to tune the `linear0`. Our overall process is summarized in the following diagram.



Currently, tune API only takes an IRModule with one `main` function, so we first get the `linear0` out into another module's main function and pass it to tune

```
mod_linear = tvm.IRModule.from_expr(MyModuleMixture["linear0"].with_attr(
↪"global_symbol", "main"))
IPython.display.HTML(code2html(mod_linear.script()))
```

```
database = ms.tune_tir(
    mod=mod_linear,
    target="llvm --num-cores=1",
    max_trials_global=64,
    num_trials_per_iter=64,
    work_dir="./tune_tmp",
)
sch = ms.tir_integration.compile_tir(database, mod_linear, "llvm --num-cores=1
↪")
```

```
2025-07-10 12:32:57 [INFO] Logging directory: ./tune_tmp/logs
2025-07-10 12:32:57 [INFO] LocalBuilder: max_workers = 10
2025-07-10 12:32:58 [INFO] LocalRunner: max_workers = 1
2025-07-10 12:32:58 [INFO] [task_scheduler.cc:159] Initializing Task #0: "main
↪"
```

```
2025-07-10 12:32:58 [DEBUG] [task_scheduler.cc:318]
 ID | Name |   FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------------
↪-------------------------
  0 | main | 200832 |      1 |            N/A |          N/A |              ␣
↪   N/A |      0 |
-------------------------------------------------------------------------------
↪-------------------------
Total trials: 0
Total latency (us): 0
```

(continues on next page)

(continued from previous page)

```
Total trials: 0
Total latency (us): 0

2025-07-10 12:32:58 [INFO] [task_scheduler.cc:180] TaskScheduler picks Task
↪#0: "main"
2025-07-10 12:33:01 [INFO] [task_scheduler.cc:193] Sending 64 sample(s) to␣
↪builder
2025-07-10 12:33:11 [INFO] [task_scheduler.cc:195] Sending 64 sample(s) to␣
↪runner
2025-07-10 12:33:23 [DEBUG] XGB iter   0: tr-p-rmse: 0.231555       tr-a-
↪peak@32: 0.999677  tr-rmse: 0.329053       tr-rmse: 0.329053
2025-07-10 12:33:24 [DEBUG] XGB iter  25: tr-p-rmse: 0.025260       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.385577       tr-rmse: 0.385577
2025-07-10 12:33:24 [DEBUG] XGB iter  50: tr-p-rmse: 0.025260       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.385580       tr-rmse: 0.385580
2025-07-10 12:33:24 [DEBUG] XGB iter  75: tr-p-rmse: 0.025260       tr-a-
↪peak@32: 1.000000  tr-rmse: 0.385580       tr-rmse: 0.385580
2025-07-10 12:33:24 [DEBUG] XGB stopped. Best iteration: [25] tr-p-rmse:0.
↪02526     tr-a-peak@32:1.00000    tr-rmse:0.38558 tr-rmse:0.38558
2025-07-10 12:33:24 [INFO] [task_scheduler.cc:237] [Updated] Task #0: "main"
```

```
2025-07-10 12:33:24 [DEBUG] [task_scheduler.cc:318]
 ID | Name |   FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------
↪-----------------------
  0 | main | 200832 |      1 |        14.2695 |      14.0742 |             ␣
↪14.0742 |     64 |
-------------------------------------------------------------------------
↪-----------------------
Total trials: 64
Total latency (us): 14.0742


Total trials: 64
Total latency (us): 14.0742

2025-07-10 12:33:24 [INFO] [task_scheduler.cc:260] Task #0 has finished.␣
↪Remaining task(s): 0
```

```
2025-07-10 12:33:24 [DEBUG] [task_scheduler.cc:318]
 ID | Name |   FLOP | Weight | Speed (GFLOPS) | Latency (us) | Weighted␣
↪Latency (us) | Trials | Done
-------------------------------------------------------------------------
↪-----------------------
  0 | main | 200832 |      1 |        14.2695 |      14.0742 |             ␣
↪14.0742 |     64 |    Y
-------------------------------------------------------------------------
↪-----------------------
Total trials: 64
Total latency (us): 14.0742
```

(continues on next page)

```
Total trials: 64
Total latency (us): 14.0742
```

Now we need to replace the original `linear0` with the new function after tuning. We can do that by first getting a `global_var`, a `pointer` reference to the functions inside the IRModule, then calling `update_func` to replace the function with the new one.

```
MyModuleWithParams2 = relax.transform.BindParams("main", nd_
↪params)(MyModuleMixture)
new_func = sch.mod["main"].with_attr("global_symbol", "linear0")
gv = MyModuleWithParams2.get_global_var("linear0")
MyModuleWithParams2.update_func(gv, new_func)
IPython.display.HTML(code2html(MyModuleWithParams2.script()))
```

We can find that the `linear0` has been replaced in the above code.

```
ex = relax.build(MyModuleWithParams2, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())

nd_res = vm["main"](data_nd)

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MyModuleWithParams2 Prediction:", class_names[pred_kind[0]])
```

```
MyModuleWithParams2 Prediction: Pullover
```

Running the code again, we can find that we get an observable amount of time reduction, mainly thanks to the new `linear0` function.

```
ftimer = vm.module.time_evaluator("main", tvm.cpu(), number=50)

print("MyModuleWithParams2 time-cost: %g ms" % (ftimer(data_nd).mean * 1000))
```

```
MyModuleWithParams2 time-cost: 0.0635652 ms
```

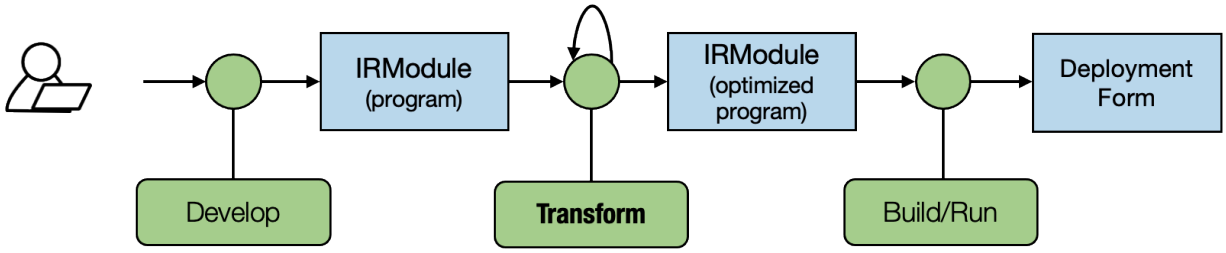
4.7 Discussions

We might notice that our previous two chapters focused on **abstraction** while this chapter starts to focus on **transformation**. Stochastic transformations specify what can be possibly optimized without nailing down all the choices. The meta-schedule API helps us to search over the space of possible transformations and pick the best one.

Importantly, putting the search result back into the end-to-end flow is just a matter of replacing the implementation of the original function with a new one that is informed by the tuning process.

So we again are following the generic MLC process in the figure below. In future lectures, we will introduce more kinds of transformations on primitive functions and computational graph functions. A good MLC process

composes these transformations together to form an end deployment form.



4.8 Summary

- Stochastic transformations help us to specify a search space of possible programs.
- MetaSchedule searches over the search space and finds an optimized one.
- We can use another transformation to replace the primitive tensor function with optimized ones and an updated end-to-end execution flow.

5 | Integration with Machine Learning Frameworks

5.1 Prelude

In the past chapters, we have learned about abstractions for machine learning compilation and transformations among tensor functions.

This chapter will discuss how to bring machine learning models from the existing ML framework into an MLC flow.

5.2 Preparations

To begin with, we will import necessary dependencies.

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

```
import torch
import torch.nn as nn
from torch import fx
from torch.nn import functional as F
```

5.3 Build an IRModule Through a Builder

In the past chapters, we have been building IRModule by directly writing TVMScript. As the model gets larger, we need a programmatical way to build up an IRModule. In this section, let us review some of the tools to support that process.

5.3.1 Tensor Expression for TensorIR Creation

First, we review the tensor expression domain-specific language to build TensorIR functions.

```
from tvm import te
```

We begin by creating a placeholder object, which represents an input to a TensorIR function.

```
A = te.placeholder((128, 128), name="A", dtype="float32")
B = te.placeholder((128, 128), name="B", dtype="float32")
```

Each input and intermediate result here are represented as a `te.Tensor` object.

```
type(A)
```

```
tvm.te.tensor.Tensor
```

Each `te.Tensor` has a shape field and dtype field that tracks the shape and data type of the computation.

```
A.shape
```

```
[128, 128]
```

We can describe computations through a sequence of tensor expression computation, Here `te.compute` takes the signature `te.compute(output_shape, fcompute)`. And the fcompute function describes how we want to compute the value of each element `[i, j]` for a given index.

The `te_matmul` function takes in an object with type `te.Tensor`, and returns the matrix multiplication result. Note how we build up computations depending on A and B's input shape. The `te_matmul` works for A and B with different input shapes.

```
def te_matmul(A: te.Tensor, B: te.Tensor) -> te.Tensor:
    assert A.shape[1] == B.shape[0]
    n = A.shape[0]
    m = B.shape[1]
    k = te.reduce_axis((0, A.shape[1]), name="k")
    return te.compute((n, m), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),␣
↪name="matmul")
```

We can create the result of matmul calling `te_matmul` with A and B.

```
C = te_matmul(A, B)
```

To create a TensorIR function, we can call `te.create_prim_func` and pass in the input and output values.

```
te.create_prim_func([A, B, C]).show()
```

We can create a tensor expression for relu computation in a similar fashion. Here we write it in a way so that `te_relu` function can work for `te.Tensor` with any dimension and shape.

```
def te_relu(A: te.Tensor) -> te.Tensor:
    return te.compute(A.shape, lambda *i: te.max(A(*i), 0), name="relu")
```

Let us try out `te_relu` on two different input shapes and dimensions. First `X1` with shape `(10,)`.

```
X1 = te.placeholder((10,), name="X1", dtype="float32")
Y1 = te_relu(X1)
te.create_prim_func([X1, Y1]).show()
```

Then `X2` with shape `(10, 20)`.

```
X2 = te.placeholder((10, 20), name="X1", dtype="float32")
Y2 = te_relu(X2)
te.create_prim_func([X2, Y2]).show()
```

One final thing that `te` API allows us to do is to compose operations and create "fused" operators. For example, we can take the result of matmul and apply relu again.

```
C = te_matmul(A, B)
D = te_relu(C)
```

We can create a TensorIR function by only passing the input and output values of interest, skipping intermediate values. This will cause the result of matmul being allocated as a temp space in the TensorIR function.

```
te.create_prim_func([A, B, D]).show()
```

We can also pass the intermediate result C into the argument list. In this case, the TensorIR function expects us to also pass in the buffer of C from the caller side. Normally we recommend only passing in the input/output so we can have more advanced fusion inside.

```
te.create_prim_func([A, B, C, D]).show()
```

5.3.2 Use BlockBuilder to Create an IRModule

So far, we have created a single TensorIR function. In order to build end-to-end model execution, we also need to be able to connect multiple TensorIR functions through a computational graph.

Let us first create a block builder, which helps us incrementally build a `relax.Function`.

```
A = relax.Var("A", relax.TensorStructInfo((128, 128), "float32"))
B = relax.Var("B", relax.TensorStructInfo((128, 128), "float32"))
```

We construct the relax function by creating a block builder and then a sequence of primitive tensor operations.

```
bb = relax.BlockBuilder()

with bb.function("main"):
    with bb.dataflow():
        C = bb.emit_te(te_matmul, A, B)
```

(continues on next page)

(continued from previous page)

```
        D = bb.emit_te(te_relu, C)
        R = bb.emit_output(D)
    bb.emit_func_output(R, params=[A, B])

MyModule = bb.get()
MyModule.show()
```

5.3.3 Deep Dive into Block Builder APIs

Now let us do a deep dive into each block builder API. It is helpful to put the block builder code and the resulting module side by side.

```
with bb.function("main"):
    with bb.dataflow():
        C = bb.emit_te(te_matmul, A, B)
        D = bb.emit_te(te_relu, C)
        R = bb.emit_output(D)
    bb.emit_func_output(R, params=[A, B])
```

```
@tvm.script.ir_module
class Module:
    @R.function
    def main(A: Tensor((128, 128), "float32"), B: Tensor((128, 128), "float32")):
        # block 0
        with R.dataflow():
            lv = R.call_tir(te_matmul, (A, B), (128, 128), dtype="float32")
            lv1 = R.call_tir(te_relu, (lv,), (128, 128), dtype="float32")
            gv: Tensor((128, 128), "float32") = lv1
            R.output(gv)
        return gv
```

The block builder comes with scopes that correspond to the scopes in the relax function. For example, `bb.dataflow()` creates a dataflow block where all the block builder calls inside the scope belonging to the dataflow scope.

```
with bb.function("main"):
    with bb.dataflow():
        # every emit call generates a variable inside a dataflow block.
```

Each intermediate result is a `relax.Var` corresponding to a variable that stores the result of the computation. `DataflowVar` indicates that the var is an intermediate step inside a dataflow block (computational graph).

```
type(C)
```

```
tvm.relax.expr.DataflowVar
```

```
isinstance(C, relax.Var)
```

```
True
```

Each line in the relax function is generated by an `emit_te` call. For example,

```
lv = R.call_dps_packed(te_matmul, (A, B), (128, 128), dtype="float32")
```

is generated by

```
C = bb.emit_te(te_matmul, A, B).
```

Under the hood, the bb.emit_te does the following things:

- Create an input `te.placeholder` for A and B
- Run them through `te_matmul` function.
- Call into `te.create_prim_func` to create a TensorIR function.
- Generate a call into the function via `call_dps_packed`.

We can find that the result is a computational graph with two intermediate values, with one node corresponding to the te_matmul operation and another one corresponding to `te_relu`.

We can create output variable of each dataflow block through `bb.emit_output`.

```
with bb.dataflow():
    ...
    R = bb.emit_output(D)
```

The above code marks that D is a variable that can be referred to outside of the dataflow block.

Finally, the function output is marked by `bb.emit_func_output`. We can only call `emit_func_output` once in each function scope.

Notably, we can specify the list of parameters of the function in the output emission stage. Doing so helps us in cases where we collect the list of parameters on the fly.

```
with bb.function("main"):
    ...
    # specify parameters in the end
    bb.emit_func_output(R, params=[A, B])
```

Alternatively, we can specify the list of parameters at the beginning of the function scope.

```
# specify parameters in the beginning.
with bb.function("main", params=[A, B]):
    ...
    bb.emit_func_output(R)
```

5.4 Import Model From PyTorch

Now that we have learned the tools to construct an IRModule programmatically. Let us use them to bring a model from PyTorch into the IRModule format.

Most machine learning framework comes with computational graph abstractions, where each node corresponds to an operation, and the edges correspond to the dependency among them. We will take a PyTorch model, obtain a computational graph in PyTorch's native format, and translate that into IRModule.

Let us begin by defining a model in PyTorch. To keep the example consistent, we will use matmul relu example.

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.weight = nn.Parameter(torch.randn(128, 128))

    def forward(self, x):
        x = torch.matmul(x, self.weight)
        x = torch.relu(x)
        return x
```

5.4.1 Create TorchFX GraphModule

We use TorchFX to trace a graph from the PyTorch module.

```
model = MyModel()
fx_module = fx.symbolic_trace(model)
type(fx_module)
```

```
torch.fx.graph_module.GraphModule.__new__.<locals>.GraphModuleImpl
```

The `fx_module` contains a simple computation graph view that can be printed as tabular data. Our goal is to translate this graph into an IRModule.

```
fx_module.graph.print_tabular()
```

```
opcode         name    target                                                  ␣
↪    args         kwargs
-------------  ------  ------------------------------------------------------
↪--  -----------  --------
placeholder    x       x                                                       ␣
↪    ()           {}
get_attr       weight  weight                                                  ␣
↪    ()           {}
call_function  matmul  <built-in method matmul of type object at␣
↪0x7af870a5c480>  (x, weight)  {}
call_function  relu    <built-in method relu of type object at 0x7af870a5c480>
↪    (matmul,)    {}
output         output  output                                                  ␣
↪    (relu,)      {}
```

5.4.2 Create Map Function

Let us define the overall high-level translation logic. The main flow is as follows:

- Create a `node_map` that maps `fx.Node` to the corresponding `relax.Var` that represents the translated node in IRModule.
- Iterate over the nodes in the fx graph in topological order.
- Compute the mapped output of the node given the mapped inputs.

```
def map_param(param: nn.Parameter):
    return relax.const(
        param.data.cpu().numpy(), relax.TensorStructInfo(param.data.shape,
↪"float32")
    )

def fetch_attr(fx_mod, target: str):
    """Helper function to fetch an attr"""
    target_atoms = target.split('.')
    attr_itr = fx_mod
    for i, atom in enumerate(target_atoms):
        if not hasattr(attr_itr, atom):
            raise RuntimeError(f"Node referenced nonexistant target {'.'.
↪join(target_atoms[:i])}")
        attr_itr = getattr(attr_itr, atom)
    return attr_itr

def from_fx(fx_mod, input_shapes, call_function_map, call_module_map):
    input_index = 0
    node_map = {}
    named_modules = dict(fx_mod.named_modules())

    bb = relax.BlockBuilder()

    fn_inputs = []
    fn_output = None
    with bb.function("main"):
        with bb.dataflow():
            for node in fx_mod.graph.nodes:
                if node.op == "placeholder":
                    # create input placeholder
                    shape = input_shapes[input_index]
                    input_index += 1
                    input_var = relax.Var(
                        node.target, relax.TensorStructInfo(shape, "float32")
                    )
                    fn_inputs.append(input_var)
                    node_map[node] = input_var
                elif node.op == "get_attr":
                    node_map[node] = map_param(fetch_attr(fx_mod, node.
↪target))
                elif node.op == "call_function":
                    node_map[node] = call_function_map[node.target](bb, node_
↪map, node)
                elif node.op == "call_module":
                    named_module = named_modules[node.target]
                    node_map[node] = call_module_map[type(named_module)](bb,␣
↪node_map, node, named_module)
                elif node.op == "output":
                    output = node_map[node.args[0]]
                    assert fn_output is None
                    fn_output = bb.emit_output(output)
        # output and finalize the function
        bb.emit_func_output(output, fn_inputs)
    return bb.get()
```

We did not define the function map in the `from_fx` function. We will supply the translation rule of each torch function via a map. Specifically, the following code block shows how we can do that through the `emit_te` API.

```
def map_matmul(bb, node_map, node: fx.Node):
    A = node_map[node.args[0]]
    B = node_map[node.args[1]]
    return bb.emit_te(te_matmul, A, B)

def map_relu(bb, node_map, node: fx.Node):
    A = node_map[node.args[0]]
    return bb.emit_te(te_relu, A)

MyModule = from_fx(
    fx_module,
    input_shapes = [(1, 128)],
    call_function_map = {
      torch.matmul: map_matmul,
      torch.relu: map_relu,
    },
    call_module_map={},
)

MyModule.show()
```

5.5 Coming back to FashionMNIST Example

```
import torch
import torchvision

test_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=1,␣
↪shuffle=True)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

img, label = next(iter(test_loader))
img = img.reshape(1, 28, 28).numpy()
```

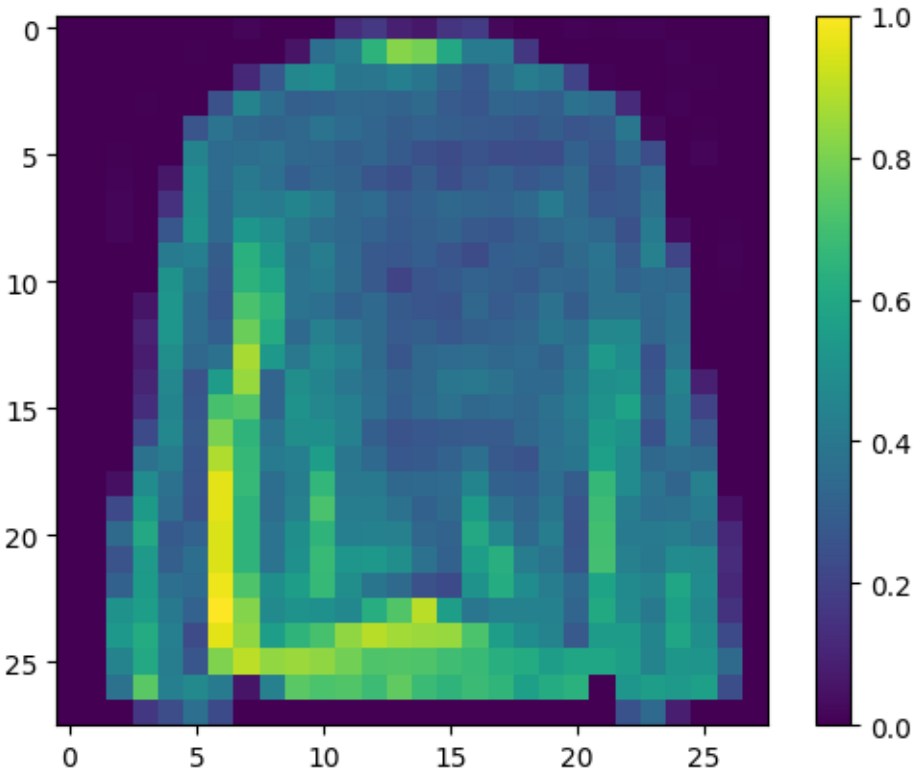
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw
```

(continues on next page)

(continued from previous page)

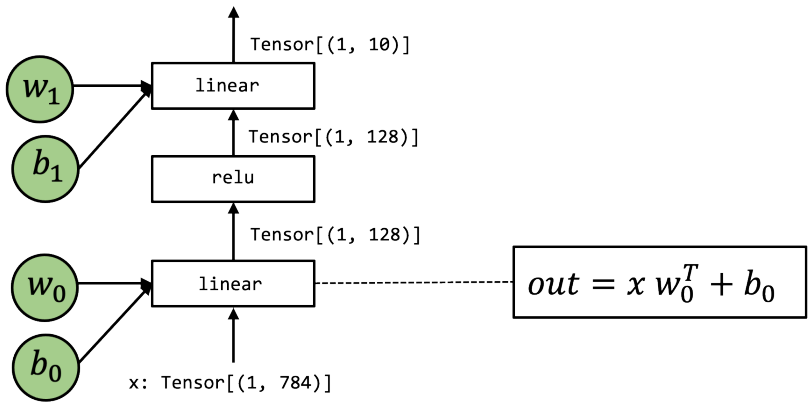
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/
↪FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
↪labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100.0%Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/
↪FashionMNIST/raw
```

```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(img[0])
plt.colorbar()
plt.grid(False)
plt.show()

print("Class:", class_names[label[0]])
```




```
Class: Pullover
```

```
# Hide outputs
!wget -nc https://github.com/mlc-ai/web-data/raw/main/models/fasionmnist_mlp_
↪params.pkl
```



The above is our model of interest, we can build the PyTorch model as follows.

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.linear0 = nn.Linear(784, 128, bias=True)
        self.relu = nn.ReLU()
```

(continues on next page)

(continued from previous page)

```
        self.linear1 = nn.Linear(128, 10, bias=True)

    def forward(self, x):
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x
```

```
import pickle as pkl

mlp_model = MLP()

mlp_params = pkl.load(open("fasionmnist_mlp_params.pkl", "rb"))
mlp_model.linear0.weight.data = torch.from_numpy(mlp_params["w0"])
mlp_model.linear0.bias.data = torch.from_numpy(mlp_params["b0"])
mlp_model.linear1.weight.data = torch.from_numpy(mlp_params["w1"])
mlp_model.linear1.bias.data = torch.from_numpy(mlp_params["b1"])
```

```
torch_res = mlp_model(torch.from_numpy(img.reshape(1, 784)))

pred_kind = np.argmax(torch_res.detach().numpy(), axis=1)
print("Torch Prediction:", class_names[pred_kind[0]])
```

```
Torch Prediction: Shirt
```

Let us try to translate from fx by defining mapping functions for the corresponding `nn.Module`. Here we are reusing pre-defined TE libraries from TVM `topi` instead of defining our own tensor expression.

- `topi.nn.dense(x, w)` performs transposed matrix multiplication `x @ w.T`
- `topi.add` performs broadcast add.

```
from tvm import topi


def map_nn_linear(bb, node_map, node, nn_mod):
    x = node_map[node.args[0]]
    w = map_param(nn_mod.weight)
    if nn_mod.bias is not None:
        b = map_param(nn_mod.bias)
    y = bb.emit_te(topi.nn.dense, x, w)
    return bb.emit_te(topi.add, y, b)

def map_nn_relu(bb, node_map, node, nn_mod):
    return map_relu(bb, node_map, node)


MLPModule = from_fx(
    fx.symbolic_trace(mlp_model),
    input_shapes = [(1, 784)],
    call_function_map={
    },
```

(continues on next page)

(continued from previous page)

```
    call_module_map={
        torch.nn.Linear: map_nn_linear,
        torch.nn.ReLU: map_nn_relu,
    },
)

MLPModule.show()
```

```
ex = relax.build(MLPModule, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())
data_nd = tvm.nd.array(img.reshape(1, 784))

nd_res = vm["main"](data_nd)

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MLPModule Prediction:", class_names[pred_kind[0]])
```

```
MLPModule Prediction: Shirt
```

5.6 Remark: Translating into High-level Operators

In most machine learning frameworks, it is sometimes helpful to first translate into high-level built-in primitive operators. The following code block gives an example to do that.

```
def map_nn_relu_op(bb, node_map, node, nn_mod):
    A = node_map[node.args[0]]
    return bb.emit(relax.op.nn.relu(A))

def map_nn_linear_op(bb, node_map, node, nn_mod):
    x = node_map[node.args[0]]
    w = map_param(nn_mod.weight)
    b = map_param(nn_mod.bias)
    return bb.emit(relax.op.linear(x, w, b))

MLPModuleHighLevel = from_fx(
    fx.symbolic_trace(mlp_model),
    input_shapes = [(1, 784)],
    call_function_map={
    },
    call_module_map={
        torch.nn.Linear: map_nn_linear_op,
        torch.nn.ReLU: map_nn_relu_op,
    },
)

MLPModuleHighLevel.show()
```

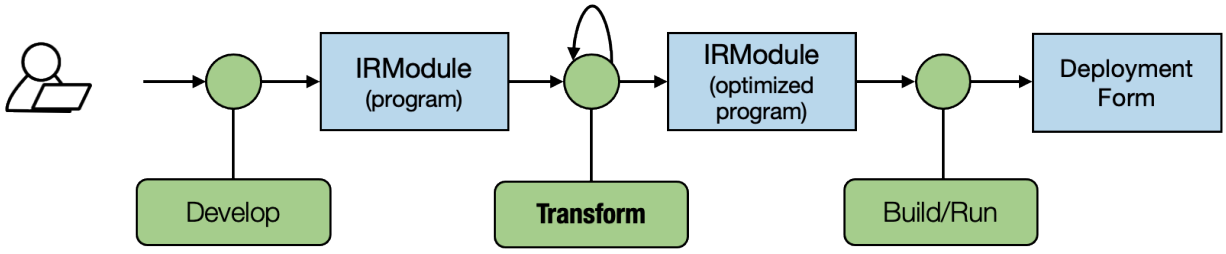
After we get the model into IRModule with those built-in operator calls. These built-in operators are **higher-level abstraction** than the TensorIR functions. There can be different opportunities to further translate these primitive operators into either library or TensorIR functions.

In most cases, it can be helpful to translate into high-level builtins when they are available. However, there are many cases where we cannot find the corresponding high-level built-in or when we want to specify the TensorIR function directly. In those cases, we can customize the translation logic or transformation to generate `call_dps_packed` or call into the library functions. Usually, we can get the best result by combining the high-level op, TensorIR, and library abstractions. We will discuss the tradeoffs in the follow-up lectures.

5.7 Discussions

In this chapter, we focus on the **develop** part of the MLC flow. We studied different ways to get models from machine learning frameworks onto the IRModule. We also briefly touched upon the high-level primitive operators.

Once we get the model into the IRModule, we can introduce more kinds of transformations on primitive functions and computational graph functions. A good MLC process composes these transformations together to form an end deployment form.



5.8 Summary

- Tensor expression API allows us to create a primitive TensorIR function.
- BlockBuilder API creates IRModule through `emit_te` and other functions.
- Integrate with existing machine learning frameworks by transforming models into an IRModule.

6 | GPU and Hardware Acceleration

6.1 Part 1

In the past chapter, we discussed MLC flows in CPU environments. This chapter will discuss how to bring some of the optimizations onto GPU. We are going to use CUDA terminology. However, the same set of concepts applies to other kinds of GPUs as well.

6.1.1 Install packages

For this course, we will use some ongoing development in TVM, which is an open-source machine learning compilation framework. We provide the following command to install a packaged version for MLC course. The particular notebook of **part 1** depends on a CUDA 11 environment.

```
python3 -m pip install mlc-ai-nightly-cu110 -f https://mlc.ai/wheels
```

NOTE: Our build system does not have GPU support yet, so part of codes will not be evaluated.

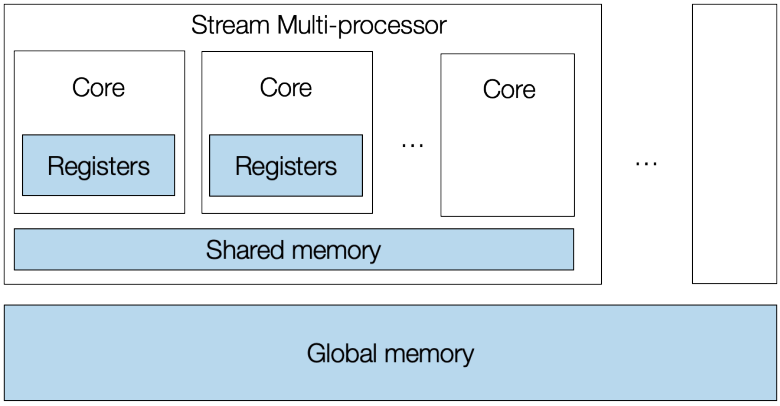
6.1.2 Preparations

To begin with, let us import the necessary dependencies.

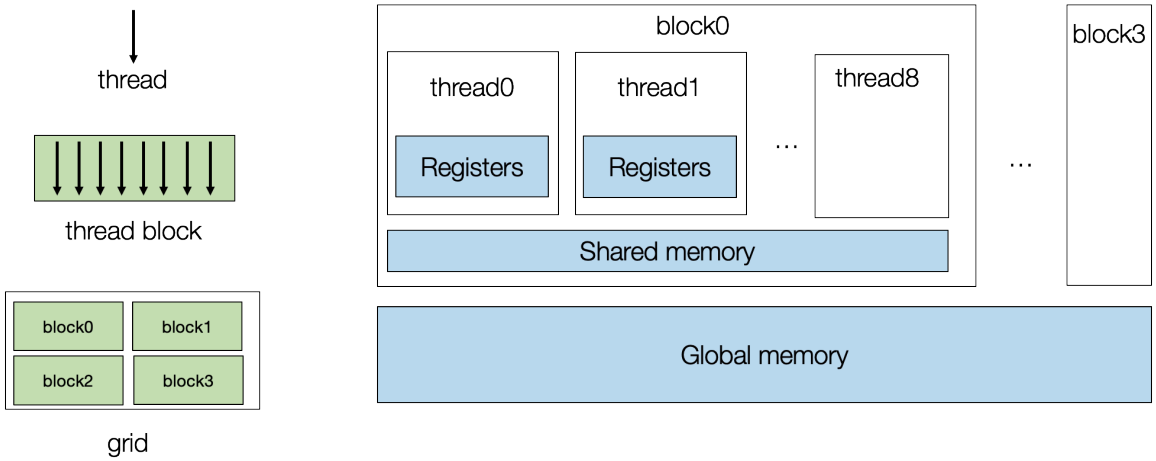
```
import numpy as np
import tvm
from tvm import relax
from tvm.ir.module import IRModule
from tvm.script import relax as R
from tvm.script import tir as T
```

6.1.3 GPU Architecture

Let us begin by reviewing what a GPU architecture looks like. A typical GPU contains a collection of stream multi-processors, and each multi-processor has many cores. A GPU device is massively parallel and allows us to execute many tasks concurrently.



To program a GPU, we need to create a set of thread blocks, with each thread mapping to the cores and the thread block map to the stream multiprocessors.



Let us start GPU programming using a vector add example. The following TensorIR program takes two vectors, A and B, performs element-wise add, and stores the result in C.

```
@tvm.script.ir_module
class MyModuleVecAdd:
    @T.prim_func
    def main(A: T.Buffer((1024,), "float32"),
             B: T.Buffer((1024,), "float32"),
             C: T.Buffer((1024,), "float32")) -> None:
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i in T.grid(1024):
            with T.block("C"):
                vi = T.axis.remap("S", [i])
                C[vi] = A[vi] + B[vi]
```

We first split loop `i` into two loops.

```
sch = tvm.tir.Schedule(MyModuleVecAdd)
block_C = sch.get_block("C")
i, = sch.get_loops(block=block_C)
```

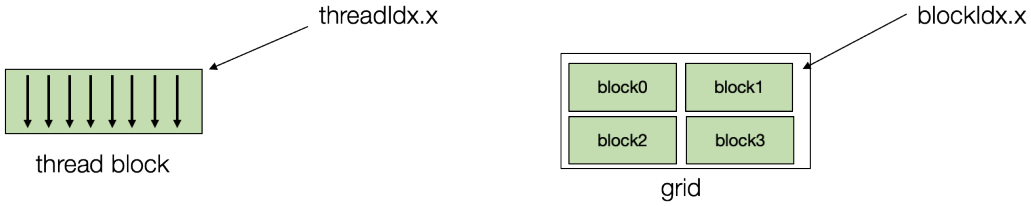
(continues on next page)

(continued from previous page)

```
i0, i1 = sch.split(i, [None, 128])
sch.mod.show()
```

GPU Thread Blocks

Then we bind the iterators to the GPU thread blocks. Each thread is parameterized by two indices – `threadIdx.x` and `blockIdx.x`. In practice, we can have multiple dimensional thread indices, but we keep them simple as one dimension.




```
sch.bind(i0, "blockIdx.x")
sch.bind(i1, "threadIdx.x")
sch.mod.show()
```

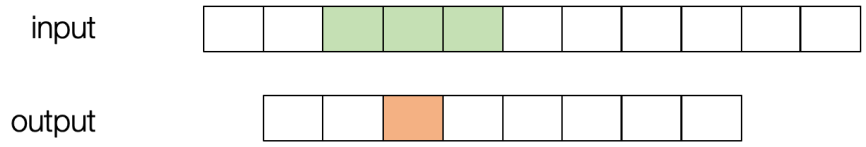
Build and Run the TensorIR Function on GPU

We can build and test out the resulting function on the GPU.

```
rt_mod = tvm.build(sch.mod, target="cuda")

A_np = np.random.uniform(size=(1024,)).astype("float32")
B_np = np.random.uniform(size=(1024,)).astype("float32")
A_nd = tvm.nd.array(A_np, tvm.cuda(0))
B_nd = tvm.nd.array(B_np, tvm.cuda(0))
C_nd = tvm.nd.array(np.zeros((1024,), dtype="float32"), tvm.cuda(0))

rt_mod["main"](A_nd, B_nd, C_nd)
print(A_nd)
print(B_nd)
print(C_nd)
```

6.1.4 Window Sum Example

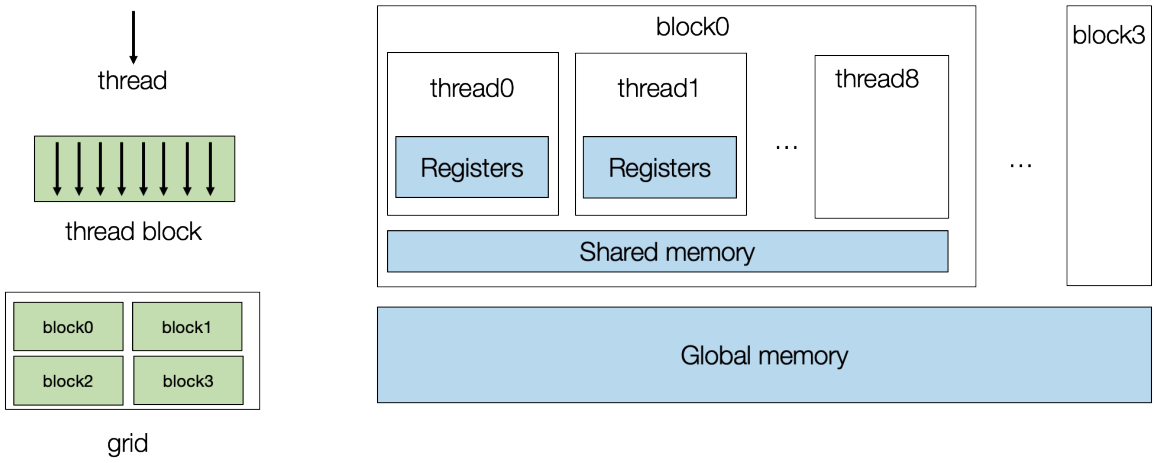
Now, let us move forward to another example – window sum. This program can be viewed as a basic version of "convolution" with a predefined weight `[1,1,1]`. We are taking sliding over the input and add three neighboring values together.




```
@tvm.script.ir_module
class MyModuleWindowSum:
    @T.prim_func
    def main(A: T.Buffer[(1027,), "float32"],
             B: T.Buffer[(1024,), "float32"]) -> None:
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i in T.grid(1024):
            with T.block("C"):
                vi = T.axis.remap("S", [i])
                B[vi] = A[vi] + A[vi + 1] + A[vi + 2]
```

First, we can bind the loop to GPU threads.

```
sch = tvm.tir.Schedule(MyModuleWindowSum)
nthread = 128
block_C = sch.get_block("C")
i,  = sch.get_loops(block=block_C)
i0, i1 = sch.split(i, [None, nthread])
sch.bind(i0, "blockIdx.x")
sch.bind(i1, "threadIdx.x")
sch.mod.show()
```



Importantly, in this case, there are reuse opportunities. Remember that each GPU thread block contains shared memory that all threads can access within the block. We use `cache_read` to add an intermediate stage that caches segments (in green below) onto the shared memory. After the caching is finished, the threads can then read from the shared memory.

```
A_shared = sch.cache_read(block_C, read_buffer_index=0, storage_scope="shared
↪")
sch.compute_at(A_shared, i1)
sch.mod.show()
```

Because the memory is shared across threads, we need to re-split the loop and bind the inner iterator of the fetching process onto the thread indices. This technique is called **cooperative fetching**, where multiple threads work together to bring the data onto the shared memory. The following reading process can be different.

```
ax = sch.get_loops(A_shared)[-1]
ax0, ax1 = sch.split(ax, [None, nthread])
sch.bind(ax1, "threadIdx.x")
sch.mod.show()
```

We can inspect the corresponding low-level code (in CUDA). The generated code contains two parts:

- A host part that calls into the GPU driver
- A cuda kernel that runs the corresponding computation.

We can print out the cuda kernel using the following code. We still need both the host and kernel code to run the program, so it is only a quick way to inspect what the final code generation result.

Notably, the build process automatically compacts the shared memory stage to use a minimum region used within the thread block.

```
rt_mod = tvm.build(sch.mod, target="cuda")
print(rt_mod.imported_modules[0].get_source())
```

Build Code for Other GPU Platforms

A MLC process usually support targeting multiple kinds of hardware platforms, we can generate Metal code(which is another kind of GPU programming model) by changing the target parameter.

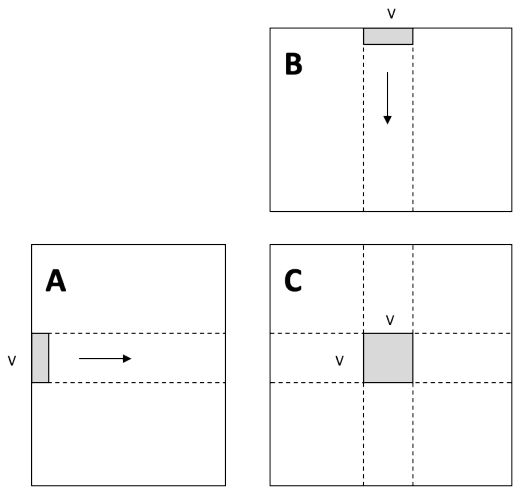
```
rt_mod = tvm.build(sch.mod, target="metal")
print(rt_mod.imported_modules[0].get_source())
```

6.1.5 Matrix Multiplication

Let us now get to something slightly more complicated and try out optimizing matrix multiplication on GPU. We will go over two common techniques for GPU performance optimization.

```
@tvm.script.ir_module
class MyModuleMatmul:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"),
             B: T.Buffer((1024, 1024), "float32"),
             C: T.Buffer((1024, 1024), "float32")) -> None:
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i, j, k in T.grid(1024, 1024, 1024):
            with T.block("C"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    C[vi, vj] = 0.0
                C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]
```

Local Blocking



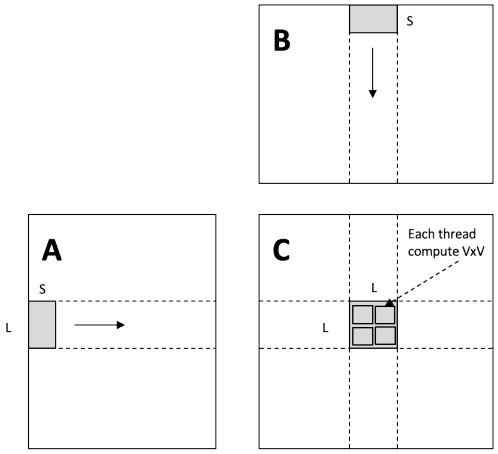
To increase overall memory reuse. We can tile the loops. In particular, we introduce local tiles such that we only need to load stripe of data from A and B once, then use them to perform a `V * V` matrix multiplication result.

This local tiling helps to reduce the memory pressure, as each element in the stripe is reused `V` times.

```
def blocking(sch,
             tile_local_y,
             tile_local_x,
             tile_block_y,
             tile_block_x,
             tile_k):
    block_C = sch.get_block("C")
    C_local = sch.cache_write(block_C, 0, "local")

    i, j, k = sch.get_loops(block=block_C)

    i0, i1, i2 = sch.split(loop=i, factors=[None, tile_block_y, tile_local_y])
    j0, j1, j2 = sch.split(loop=j, factors=[None, tile_block_x, tile_local_x])
    k0, k1 = sch.split(loop=k, factors=[None, tile_k])
    sch.unroll(k1)
    sch.reorder(i0, j0, i1, j1, k0, k1, i2, j2)
    sch.reverse_compute_at(C_local, j1)

    sch.bind(i0, "blockIdx.y")
    sch.bind(j0, "blockIdx.x")

    sch.bind(i1, "threadIdx.y")
    sch.bind(j1, "threadIdx.x")
    sch.decompose_reduction(block_C, k0)

    return sch

sch = tvm.tir.Schedule(MyModuleMatmul)
sch = blocking(sch, 8, 8, 8, 8, 4)
sch.mod.show()
```

```
rt_mod = tvm.build(sch.mod, target="cuda")
dev = tvm.cuda(0)
A_np = np.random.uniform(size=(1024, 1024)).astype("float32")
B_np = np.random.uniform(size=(1024, 1024)).astype("float32")
A_nd = tvm.nd.array(A_np, dev)
B_nd = tvm.nd.array(B_np, dev)
C_nd = tvm.nd.array(np.zeros((1024, 1024), dtype="float32"), dev)

num_flop = 2 * 1024 * 1024 * 1024
evaluator = rt_mod.time_evaluator("main", dev, number=10)

print("GEMM-Blocking: %f GFLOPS" % (num_flop / evaluator(A_nd, B_nd, C_nd).
↪mean / 1e9))
```

6.1.6 Shared Memory Blocking



Our first attempt did not consider the neighboring threads which sit in the same GPU thread block, and we can load the data they commonly need into a piece of shared memory.

The following transformation does that.

```
def cache_read_and_coop_fetch(sch, block, nthread, read_idx, read_loc):
    read_cache = sch.cache_read(block=block, read_buffer_index=read_idx,␣
↪storage_scope="shared")
    sch.compute_at(block=read_cache, loop=read_loc)
    # vectorized cooperative fetch
    inner0, inner1 = sch.get_loops(block=read_cache)[-2:]
    inner = sch.fuse(inner0, inner1)
    _, tx, vec = sch.split(loop=inner, factors=[None, nthread, 4])
    sch.vectorize(vec)
    sch.bind(tx, "threadIdx.x")


def blocking_with_shared(
    sch,
```

(continues on next page)

(continued from previous page)

```
    tile_local_y,
    tile_local_x,
    tile_block_y,
    tile_block_x,
    tile_k):
    block_C = sch.get_block("C")
    C_local = sch.cache_write(block_C, 0, "local")

    i, j, k = sch.get_loops(block=block_C)

    i0, i1, i2 = sch.split(loop=i, factors=[None, tile_block_y, tile_local_y])
    j0, j1, j2 = sch.split(loop=j, factors=[None, tile_block_x, tile_local_x])
    k0, k1 = sch.split(loop=k, factors=[None, tile_k])

    sch.reorder(i0, j0, i1, j1, k0, k1, i2, j2)
    sch.reverse_compute_at(C_local, j1)

    sch.bind(i0, "blockIdx.y")
    sch.bind(j0, "blockIdx.x")

    tx = sch.fuse(i1, j1)
    sch.bind(tx, "threadIdx.x")
    nthread = tile_block_y * tile_block_x
    cache_read_and_coop_fetch(sch, block_C, nthread, 0, k0)
    cache_read_and_coop_fetch(sch, block_C, nthread, 1, k0)
    sch.decompose_reduction(block_C, k0)

    return sch

sch = tvm.tir.Schedule(MyModuleMatmul)
sch = blocking_with_shared(sch, 8, 8, 8, 8, 8)
sch.mod.show()
```

```
rt_mod = tvm.build(sch.mod, target="cuda")
dev = tvm.cuda(0)
evaluator = rt_mod.time_evaluator("main", dev, number=10)

print("GEMM-Blocking: %f GFLOPS" % (num_flop / evaluator(A_nd, B_nd, C_nd).
↪mean / 1e9))
```

6.1.7 Leveraging Automatic Program Optimization

So far, we have been manually writing transformations to optimize the TensorIR program on GPU. We can leverage the automatic program optimization framework to tune the same program. The following code does that, we only set a small number here, and it can take a few min to finish.

```
from tvm import meta_schedule as ms

database = ms.tune_tir(
    mod=MyModuleMatmul,
    target="nvidia/tesla-p100",
    max_trials_global=64,
```

(continues on next page)

(continued from previous page)

```
    num_trials_per_iter=64,
    work_dir="./tune_tmp",
)
sch = ms.tir_integration.compile_tir(database, MyModuleMatmul, "nvidia/tesla-
↪p100")
sch.mod.show()
```

```
rt_mod = tvm.build(sch.mod, target="nvidia/tesla-p100")
dev = tvm.cuda(0)
evaluator = rt_mod.time_evaluator("main", dev, number=10)

print("MetaSchedule: %f GFLOPS" % (num_flop / evaluator(A_nd, B_nd, C_nd).
↪mean / 1e9))
```

6.1.8 Summary

This chapter studies another axis of MLC – how we can transform our program for hardware acceleration. The MLC process helps us to bridge the input models toward different GPU programming models and environments. We will visit more hardware specialization topics in the incoming chapter as well.

- A typical GPU contains two-level hierarchy. Each thread is indexed by(in cuda terminology) `threadIdx.x` and `blockIdx.x`(there can be multiple dimension indices as well, but they can be fused to one.
- Shared memory helps cache data commonly used across the threads within the same block.
- Encourage memory reuse during GPU optimization.

6.2 Part 2

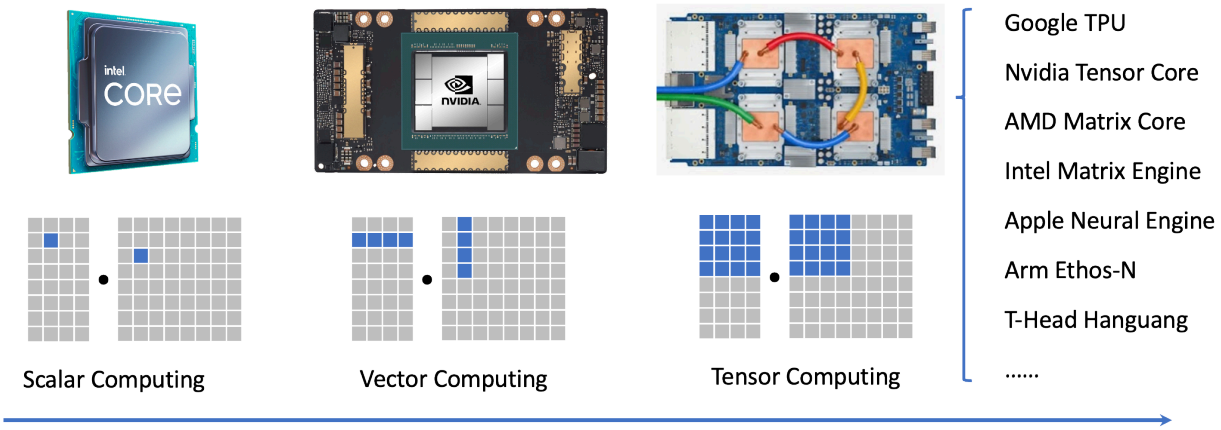
We discussed building MLC flows for CPU and GPU environments in the past chapters. This chapter focuses on how we build conceptual programming models for specialized hardware backends.

6.2.1 Preparations

To begin with, let us import the necessary dependencies.

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

6.2.2 Hardware Specialization Trend



If we look at the machine learning hardware landscape, one emerging theme recently is specialization. Traditionally, we build our solutions on generic scalar processors, where we can perform operations on one floating point at a time. The vector instructions set such as AVX and ARM/Neon provide effective ways to speed up our programs but also bring some complexities to how we write the programs.

The latest accelerators for machine learning introduced specialized units for tensor computing, with instructions for multi-dimensional data copy and matrix/tensor computations.

Key Elements of Specialized Code

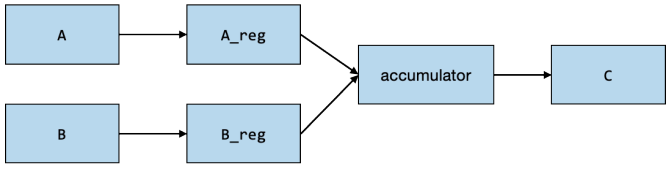
To help us better understand elements of specialized hardware programming, let us first study the following **low-level NumPy** code. While this code still runs in python, it resembles a set of possible operations that can happen in a specialized hardware backend.

```
def accel_fill_zero(C):
    C[:] = 0

def accel_tmm_add(C, A, B):
    C[:] += A @ B.T

def accel_dma_copy(reg, dram):
    reg[:] = dram[:]

def lnumpy_tmm(A: np.ndarray, B: np.ndarray, C: np.ndarray):
    # a special accumulator memory
    C_accumulator = np.empty((16, 16), dtype="float32")
    A_reg = np.empty((16, 16), dtype="float32")
    B_reg = np.empty((16, 16), dtype="float32")

    for i in range(64):
        for j in range(64):
            accel_fill_zero(C_accumulator[:,:])
            for k in range(64):
                accel_dma_copy(A_reg[:], A[i * 16 : i * 16 + 16, k * 16 : k *␣
↪16 + 16])
```

(continues on next page)

(continued from previous page)

```
                accel_dma_copy(B_reg[:], B[j * 16 : j * 16 + 16, k * 16 : k *␣
↪16 + 16])
                accel_tmm_add(C_accumulator[:,:], A_reg, B_reg)
            accel_dma_copy(C[i * 16 : i * 16 + 16, j * 16 : j * 16 + 16], C_
↪accumulator[:,:])
```



The above low-level NumPy program contains the following key elements:

- The basic unit of computation is a 16x16x16 matrix multiplication (`accel_tmm_add`)
- `accel_tmm_add` takes in two inputs – `A_reg` and `B_reg` and accumulates into an accumulator memory.
- The data copy is performed using a special function (`accel_dma_copy`).

In a real-world hardware backend, we usually expect `A_reg`, `B_reg`, and `C_accumulator` to map to special memory regions (or registers) in the hardware. These are called **special memory scopes**. Additionally, there is a limited set of hardware-accelerated operations we can perform on these settings. Operations such `accel_tmm_add` can be mapped to real hardware instructions or an efficient kernel function implementation provided by the vendor.

We can run the following code block to confirm the low-level NumPy code runs correctly.

```
dtype = "float32"
a_np = np.random.rand(1024, 1024).astype(dtype)
b_np = np.random.rand(1024, 1024).astype(dtype)
c_tmm = a_np @ b_np.T
```

```
c_np = np.empty((1024, 1024), dtype="float32")
lnumpy_tmm(a_np, b_np, c_np)
np.testing.assert_allclose(c_np, c_tmm, rtol=1e-5)
```

A Block with Tensorized Computation

One of our key observations is that the specialized accelerator code is not structured in the unit of scalar computations. Most of the TensorIR code we have run so far contains a block that computes a single element in the output Tensor. Many specialized accelerators run computations over regions of tensors. The block construct in TensorIR helps us to group such relevant computation.

```
@tvm.script.ir_module
class MatmulBlockModule:
    @T.prim_func
    def main(
```

(continues on next page)

(continued from previous page)

```
        A: T.Buffer((1024, 1024), "float32"),
        B: T.Buffer((1024, 1024), "float32"),
        C: T.Buffer((1024, 1024), "float32"),
    ) -> None:
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i0, j0, k0 in T.grid(64, 64, 64):
            with T.block("tmm-16x16"):
                vi0, vj0, vk0 = T.axis.remap("SSR", [i0, j0, k0])
                with T.init():
                    for i1, j1 in T.grid(16, 16):
                        with T.block("tmm_init"):
                            vi1, vj1 = T.axis.remap("SS", [i1, j1])
                            C[vi0 * 16 + vi1, vj0 * 16 + vj1] = T.float32(0)

                for i1, j1, k1 in T.grid(16, 16, 16):
                    with T.block("tmm"):
                        vi1, vj1, vk1 = T.axis.remap("SSR", [i1, j1, k1])
                        C[vi0 *16 + vi1, vj0 * 16 + vj1] += \
                            A[vi0 * 16 + vi1, vk0 * 16 + vk1] * B[vj0 * 16 +␣
↪vj1, vk0 * 16 + vk1]
```

```
MatmulBlockModule.show()
```

Let us take a closer look at the following block

```
with T.block("tmm-16x16"):
    T.reads(A[vi0 * 16 : vi0 * 16 + 16, vk0 * 16 : vk0 * 16 + 16], B[vj0 * 16␣
↪: vj0 * 16 + 16, vk0 * 16 : vk0 * 16 + 16])
    T.writes(C[vi0 * 16 : vi0 * 16 + 16, vj0 * 16 : vj0 * 16 + 16])
    ...
```

This block reads from a 16x16 region from `A` and `B`, and writes to a 16x16 region of `C`. In this case the content of the block contains further details about a specific implementation of the subregion computations. We call this block a **tensorized block** as they contain computations that span over sub-regions of tensors.

We can run the following code to confirm that the TensorIR module produces the correct result.

```
a_nd = tvm.nd.array(a_np)
b_nd = tvm.nd.array(b_np)

c_nd = tvm.nd.empty((1024, 1024), dtype="float32")

lib = tvm.build(MatmulBlockModule, target="llvm")
lib["main"](a_nd, b_nd, c_nd)
np.testing.assert_allclose(c_nd.numpy(), c_tmm, rtol=1e-5)
```

Transforming Loops Around Tensorized Block

One thing that we can do here is to transform the loops surrounding the tensor computation block. These loop transformations can help us to reorganize the surrounding iterations to enable a space of different tensor program variants.

```
sch = tvm.tir.Schedule(MatmulBlockModule)

block_mm = sch.get_block("tmm-16x16")
i, j, k = sch.get_loops(block_mm)

i0, i1 = sch.split(i, [None, 4])

sch.reorder(i0, j, i1, k)
sch.mod.show()
```

Blockization – Creating Tensorized Blocks

In most settings, we start with loops that come with scalar computations. TensorIR provides a primitive call blockization to group subregions of a loop together to form a tensorized computation block.

```
@tvm.script.ir_module
class MatmulModule:
    @T.prim_func
    def main(
        A: T.Buffer((1024, 1024), "float32"),
        B: T.Buffer((1024, 1024), "float32"),
        C: T.Buffer((1024, 1024), "float32"),
    ) -> None:
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        for i, j, k in T.grid(1024, 1024, 1024):
            with T.block("matmul"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    C[vi, vj] = T.float32(0)
                C[vi, vj] += A[vi, vk] * B[vj, vk]
```

```
sch = tvm.tir.Schedule(MatmulModule)
i, j, k = sch.get_loops("matmul")
i, ii = sch.split(i, factors=[None, 16])
j, ji = sch.split(j, factors=[None, 16])
k, ki = sch.split(k, factors=[None, 16])
sch.reorder(i, j, k, ii, ji, ki)
sch.mod.show()
```

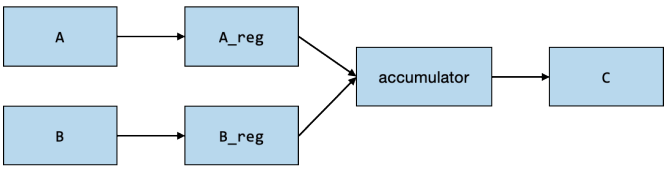
```
block_mm = sch.blockize(ii)
sch.mod.show()
```

Transforming TensorIR to Introduce Special Memory Scope

As we noted in the low-level NumPy code, one key element of the low-level TensorIR is the special memory scope used during the acceleration.

We can use cache_read and write to create intermediate memory stages.

```
A_reg = sch.cache_read(block_mm, 0, storage_scope="global.A_reg")
B_reg = sch.cache_read(block_mm, 1, storage_scope="global.B_reg")
sch.compute_at(A_reg, k)
sch.compute_at(B_reg, k)

write_back_block = sch.cache_write(block_mm, 0, storage_scope="global.
↪accumulator")
sch.reverse_compute_at(write_back_block, j)
sch.mod.show()
```



Here `global.A_reg` contains two parts. `global` indicates that all threads can globally access the memory, and `A_reg` is a **scope tag** of the memory, which provides opportunities for follow-up compilation to map it to special regions such as registers.

6.2.3 Tensorization

Now we have created a set of blocks that maps to the corresponding stages of computation in the TensorIR. The remaining step is to map some of the tensorized blocks to use a specific implementation that maps to the hardware accelerated instructions. This mapping process is called **tensorization**.

To prepare for tensorization, we first register a tensor intrinsic (TensorIntrin) that contains a description of the computation and implementation.

The system will use the description to find relevant regions that match the computation, while implementation maps the computation to accelerated hardware instructions.

```
@T.prim_func
def tmm16_desc(a: T.handle, b: T.handle, c: T.handle) -> None:
    A = T.match_buffer(a, (16, 16), "float32", offset_factor=16, scope=
↪"global.A_reg")
    B = T.match_buffer(b, (16, 16), "float32", offset_factor=16, scope=
↪"global.B_reg")
    C = T.match_buffer(c, (16, 16), "float32", offset_factor=16,  scope=
↪"global.accumulator")

    with T.block("root"):
        T.reads(C[0:16, 0:16], A[0:16, 0:16], B[0:16, 0:16])
        T.writes(C[0:16, 0:16])
```

(continues on next page)

(continued from previous page)

```
        for i, j, k in T.grid(16, 16, 16):
            with T.block(""):
                vii, vjj, vkk = T.axis.remap("SSR", [i, j, k])
                C[vii, vjj] = C[vii, vjj] + A[vii, vkk] * B[vjj, vkk]


@T.prim_func
def tmm16_impl(a: T.handle, b: T.handle, c: T.handle) -> None:
    sa = T.int32()
    sb = T.int32()
    sc = T.int32()
    A = T.match_buffer(a, (16, 16), "float32", offset_factor=16, strides=[sa,␣
↪1], scope="global.A_reg")
    B = T.match_buffer(b, (16, 16), "float32", offset_factor=16, strides=[sb,␣
↪1], scope="global.B_reg")
    C = T.match_buffer(c, (16, 16), "float32", offset_factor=16, strides=[sc,␣
↪1], scope="global.accumulator")

    with T.block("root"):
        T.reads(C[0:16, 0:16], A[0:16, 0:16], B[0:16, 0:16])
        T.writes(C[0:16, 0:16])
        T.evaluate(
            T.call_extern(
                "tmm16",
                C.access_ptr("w"),
                A.access_ptr("r"),
                B.access_ptr("r"),
                sa,
                sb,
                sc,
                dtype="int32",
            )
        )

tvm.tir.TensorIntrin.register("tmm16", tmm16_desc, tmm16_impl)
```

As a preparation step, we first decompose the reduction into an initialization block and an update step.

```
sch.decompose_reduction(block_mm, k)
sch.mod.show()
```

Then we can call tensorize, to map the `block_mm` (which corresponds to the `matmul_o_update` block) to use the implementation of `tmm16`.

```
sch.tensorize(block_mm, "tmm16")
```

```
sch.mod.show()
```

Here we use `T.call_extern` to call into an external function inside the environment. The downstream compilation step can easily map the implementation to an instruction that implements the operation.

Alternatively, we can map tmm16 to a micro-kernel that implements this tensorized computation. The following code shows the how to do that through an extern "C" code (which allows further embedding of inline

assembly if necessary).

```
def tmm_kernel():
    cc_code = """
      extern "C" int tmm16(float *cc, float *aa, float *bb, int stride_a, int␣
↪stride_b, int stride_c) {
        for (int i = 0; i < 16; ++i) {
            for (int j = 0; j < 16; ++j) {
                for (int k = 0; k < 16; ++k) {
                    cc[i * stride_c + j] += aa[i * stride_a + k] * bb[j *␣
↪stride_b + k];
                }
            }
        }
        return 0;
      }
    """
    from tvm.contrib import clang, utils

    temp = utils.tempdir()
    ll_path = temp.relpath("temp.ll")
    # Create LLVM ir from c source code
    ll_code = clang.create_llvm(cc_code, output=ll_path)
    return ll_code

sch.annotate(i, "pragma_import_llvm", tmm_kernel())
```

We can then go and execute the following code-block, which redirects the tensorized computation to the custom defined `tmm_kernel`.

```
<!-- todo -->
<!-- For CI, do not run this part of the code -->
a_nd = tvm.nd.array(a_np)
b_nd = tvm.nd.array(b_np)

c_nd = tvm.nd.empty((1024, 1024), dtype="float32")

lib = tvm.build(sch.mod, target="llvm")
lib["main"](a_nd, b_nd, c_nd)
np.testing.assert_allclose(c_nd.numpy(), c_tmm, rtol=1e-5)
```

6.2.4 Discussions

This section covers a set of key elements of specialized hardware support. One of the key constructs here is the tensorized block and computation alongside tensor subregions. TensorIR also contains additional properties that build on top of the foundational elements:

- Layout constraints in the specialized memory.
- Interaction with thread hierarchies.

We don't have enough time to cover these in one lecture, but we will add optional readings on some of the additional content.

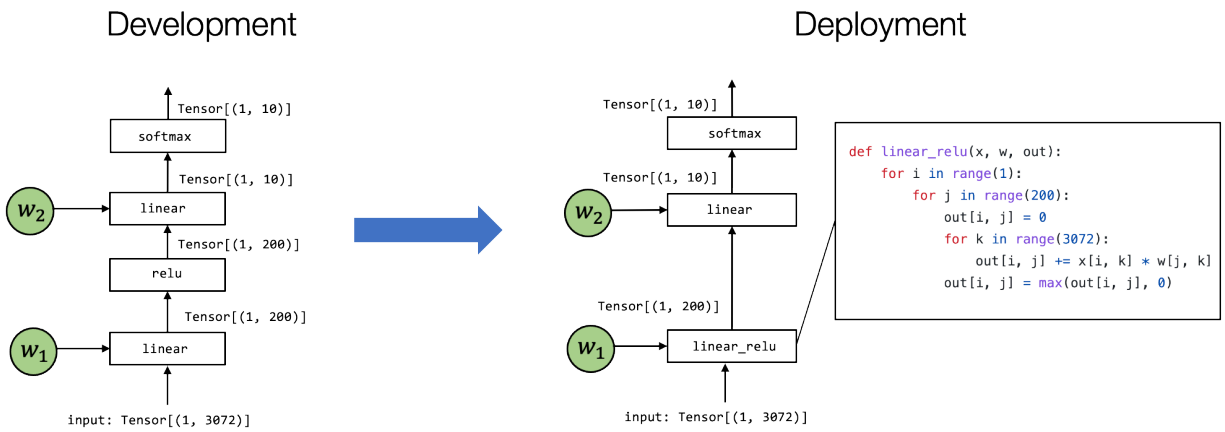
6.2.5 Summary

- Overall trend of Hardware Specialization toward tensorized computation.
- TensorIR transformations with tensorized blocks.
- Tensorization: the process of mapping block of loop computations to specialized implementations.

7 | Computational Graph Optimization

7.1 Prelude

Most of the MLC process can be viewed as transformation among tensor functions. In the past chapters, we studied how to transform each primitive tensor functions individually. In this chapter, let us talk about high-level transformations among computational graphs.



7.2 Preparations

To begin with, let us import the necessary dependencies.

```
# This is needed for deferring annotation parsing in TVMScript
import numpy as np
import tvm
from tvm import relax, topi
from tvm.ir.module import IRModule
from tvm.script import relax as R
from tvm.script import tir as T
```

7.3 Pattern Match and Rewriting

To begin with, let us start with the following example.

```
@tvm.script.ir_module
class MyModule:
    @R.function
    def main(x: R.Tensor((3, 4), "float32"), y: R.Tensor((3, 4), "float32")):
        with R.dataflow():
            lv0 = relax.op.multiply(x, y)
            gv0 = relax.op.add(lv0, y)
            R.output(gv0)
        return gv0
```

`MyModule` contains a relax function with two high-level operators, relax.op.multiply and relax.op.add. Our goal is to find these two operators and replace it with a call into `relax.op.ewise_fma` operator.

Before we dive into how to do that exactly, let us first examine the data structure that makes up the MyModule. Each IRModule contains a collection of functions, and the function body is composed of a set of data structures called abstract syntax trees (AST).

```
relax_func = MyModule["main"]
```

Each function is represented by a `relax.expr.Function` node.

```
type(relax_func)
```

```
tvm.relax.expr.Function
```

The function contains a list of parameters.

```
relax_func.params
```

```
[x, y]
```

The function contains a body fields that represents its return value and set of binding blocks in the function.

```
func_body = relax_func.body
type(func_body)
```

```
tvm.relax.expr.SeqExpr
```

The function body SeqExpr contains a sequence of (binding) blocks

```
func_body.blocks
```

```
[x: R.Tensor((3, 4), dtype="float32")
y: R.Tensor((3, 4), dtype="float32")
with R.dataflow():
```

(continues on next page)

(continued from previous page)

```
    lv0: R.Tensor((3, 4), dtype="float32") = R.multiply(x, y)
    gv0: R.Tensor((3, 4), dtype="float32") = R.add(lv0, y)
    R.output(gv0)]
```

```
dataflow_block = func_body.blocks[0]
```

In our particular case, we have a single data flow block that contains two bindings. Each binding corresponds to one of the following two lines

```
lv0 = relax.op.multiply(x, y)
gv0 = relax.op.add(lv0, y)
```

```
dataflow_block.bindings
```

```
[x: R.Tensor((3, 4), dtype="float32")
y: R.Tensor((3, 4), dtype="float32")
lv0: R.Tensor((3, 4), dtype="float32") = R.multiply(x, y), lv0: R.Tensor((3,␣
↪4), dtype="float32")
y: R.Tensor((3, 4), dtype="float32")
gv0: R.Tensor((3, 4), dtype="float32") = R.add(lv0, y)]
```

```
binding = dataflow_block.bindings[0]
```

Each binding have a var field that corresponds to the left hand side of the binding (`lv0`, `gv0`).

```
binding.var
```

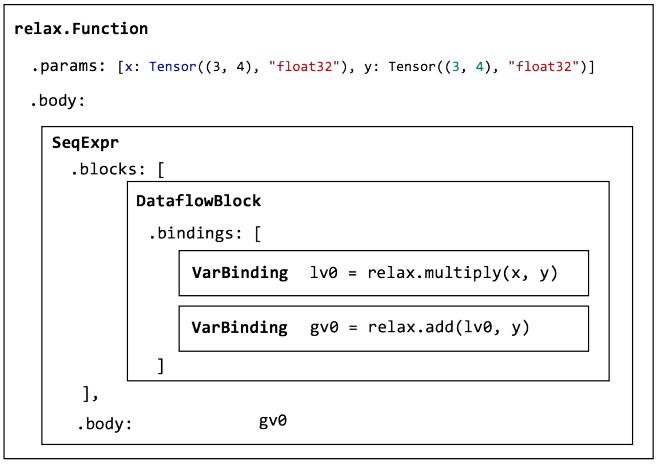
```
lv0
```

And its value field corresponds to the right-hand side of the binding. Each value field corresponds to a `relax.Call` node representing a call into a primitive function.

```
binding.value
```

```
R.multiply(x, y)
```

```
@tvm.script.ir_module
class MyModule:
@R.function
def main(x: Tensor((3, 4), "float32"),
         y: Tensor((3, 4), "float32")):
    with relax.dataflow():
        lv0 = relax.multiply(x, y)
        gv0 = relax.add(lv0, y)
        relax.output(gv0)
     return gv0
```



The above figure summarizes the data structure involved in this particular function.

One approach to rewrite the program would be to traverse MyModule's AST recursively and generate a transformed AST. We can certainly do that using the python API available. However, we can use extra tooling support to simplify the process. The following code block follows a design pattern called **visitor pattern** that allows us to visit each AST node and rewrite them to transformed versions.

```
@relax.expr_functor.mutator
class EwiseFMARewriter(relax.PyExprMutator):
    def visit_call_(self, call):
        call = self.visit_expr_post_order(call)
        add_op = tvm.ir.Op.get("relax.add")
        multiply_op = tvm.ir.Op.get("relax.multiply")
        ewise_fma_op = tvm.ir.Op.get("relax.ewise_fma")

        if call.op != add_op:
            return call

        value = self.lookup_binding(call.args[0])
        if not isinstance(value, relax.Call) or value.op != multiply_op:
            return call

        fma_call = relax.Call(
            ewise_fma_op, [value.args[0], value.args[1], call.args[1]], None,␣
↪None
        )
        return fma_call


updated_fn = EwiseFMARewriter().visit_expr(MyModule["main"])
updated_fn.show()
```

We can go ahead and run the code. Note that the result rewrites gv0 to the fused operator but leaves lv0 in the code. We can use `remove_all_unused` to further simplify the code block.

```
relax.analysis.remove_all_unused(updated_fn).show()
```

7.4 Fuse Linear and ReLU

Now we have get a basic taste of graph rewriting. Let us try it on an end to end model.

```
# Hide outputs
!wget https://github.com/mlc-ai/web-data/raw/main/models/fasionmnist_mlp_
↪params.pkl
```

```
import pickle as pkl

mlp_params = pkl.load(open("fasionmnist_mlp_params.pkl", "rb"))
```

The following code reconstructs the FashionMNIST MLP model we used in our past chapters. To simplify our explaination, we directly construct the model using high-level operators such as `relax.op.add` and `relax.op.matmul`.

```
def create_model():
    bb = relax.BlockBuilder()
    x = relax.Var("x", relax.TensorStructInfo((1, 784), "float32"))
    w0 = relax.const(mlp_params["w0"], "float32")
    b0 = relax.const(mlp_params["b0"], "float32")
    w1 = relax.const(mlp_params["w1"], "float32")
    b1 = relax.const(mlp_params["b1"], "float32")
    with bb.function("main", [x]):
        with bb.dataflow():
            lv0 = bb.emit(relax.op.matmul(x, relax.op.permute_dims(w0)))
            lv1 = bb.emit(relax.op.add(lv0, b0))
            lv2 = bb.emit(relax.op.nn.relu(lv1))
            lv3 = bb.emit(relax.op.matmul(lv2, relax.op.permute_dims(w1)))
            lv4 = bb.emit(relax.op.add(lv3, b1))
            gv = bb.emit_output(lv4)
        bb.emit_func_output(gv)

    return bb.get()

MLPModel = create_model()
MLPModel.show()
```

We aim to "fuse" the dense and add operations into a single group. The following code achieves that through the following steps:

- Identify `matmul` and `add` patterns.
- Generate another fused sub-function that calls into the matmul and add operators.
- Replace `matmul` and `add` with the fused sub-functions.

```
@relax.expr_functor.mutator
class MatmulAddFusor(relax.PyExprMutator):
    def __init__(self, mod: IRModule) -> None:
        super().__init__()
        self.mod_ = mod
        # cache pre-defined ops
        self.add_op = tvm.ir.Op.get("relax.add")
```

(continues on next page)

(continued from previous page)

```
        self.matmul_op = tvm.ir.Op.get("relax.matmul")
        self.counter = 0

    def transform(self) -> IRModule:
        for global_var, func in self.mod_.functions.items():
            if not isinstance(func, relax.Function):
                continue
            # avoid already fused primitive functions
            if func.attrs is not None and "Primitive" in func.attrs.keys()␣
↪and func.attrs["Primitive"] != 0:
                continue
            updated_func = self.visit_expr(func)
            updated_func = relax.analysis.remove_all_unused(updated_func)
            self.builder_.update_func(global_var, updated_func)

        return self.builder_.get()

    def visit_call_(self, call):
        call = self.visit_expr_post_order(call)

        def match_call(node, op):
            if not isinstance(node, relax.Call):
                return False
            return node.op == op

        # pattern match matmul => add
        if not match_call(call, self.add_op):
            return call

        value = self.lookup_binding(call.args[0])
        if value is None:
            return call

        if not match_call(value, self.matmul_op):
            return call

        x = value.args[0]
        w = value.args[1]
        b = call.args[1]

        # construct a new fused primitive function
        param_x = relax.Var("x" ,relax.TensorStructInfo(x.struct_info.shape,␣
↪x.struct_info.dtype))
        param_w = relax.Var("w" ,relax.TensorStructInfo(w.struct_info.shape,␣
↪w.struct_info.dtype))
        param_b = relax.Var("b" ,relax.TensorStructInfo(b.struct_info.shape,␣
↪b.struct_info.dtype))

        bb = relax.BlockBuilder()

        fn_name = "fused_matmul_add%d" % (self.counter)
        self.counter += 1
        with bb.function(fn_name, [param_x, param_w, param_b]):
            with bb.dataflow():
                lv0 = bb.emit(relax.op.matmul(param_x, param_w))
```

(continues on next page)

(continued from previous page)

```
                gv = bb.emit_output(relax.op.add(lv0, param_b))
            bb.emit_func_output(gv)

        # Add Primitive attribute to the fused funtions
        fused_fn = bb.get()[fn_name].with_attr("Primitive", 1)
        global_var = self.builder_.add_func(fused_fn, fn_name)

        # construct call into the fused function
        return relax.Call(global_var, [x, w, b], None, None)

@tvm.ir.transform.module_pass(opt_level=2, name="MatmulAddFuse")
class FuseDenseAddPass:
    """The wrapper for the LowerTensorIR pass."""
    def transform_module(self, mod, ctx):
        return MatmulAddFusor(mod).transform()


MLPFused = FuseDenseAddPass()(MLPModel)
MLPFused.show()
```

7.4.1 Why Creating a Sub-function

In the above example, we created two sub-functions with the prefix `fuse_matmul_add`. These sub-function bodies contain information about the operations performed by the fused operator. An alternative to this rewriting is simply creating a separate primitive operation for the fused operator (like `ewise_fma`). However, as we are looking into fusing more operators, there can be an exponential amount of possible combinations. A sub-function that groups the fused operation together provides the same amount of information for follow-up code lowering without introducing a dedicated high-level operator for each fusion pattern.

7.5 Map to TensorIR Calls

The fused IRModule only contains calls into high-level operations. To further low-level optimization and code generation, we need to translate those high-level primitive operators into corresponding TensorIR functions (or environment library functions).

The following code remaps high-level operations to the corresponding TensorIR functions. Here we leverage the internal block builder in each Mutator and return the transformed value using `call_te`.

```
@relax.expr_functor.mutator
class LowerToTensorIR(relax.PyExprMutator):
    def __init__(self, mod: IRModule, op_map) -> None:
        super().__init__()
        self.mod_ = mod
        self.op_map = {
            tvm.ir.Op.get(k): v for k, v in op_map.items()
        }


    def visit_call_(self, call):
        call = self.visit_expr_post_order(call)
```

(continues on next page)

(continued from previous page)

```
        if call.op in self.op_map:
            return self.op_map[call.op](self.builder_, call)
        return call

    def transform(self) -> IRModule:
        for global_var, func in self.mod_.functions.items():
            if not isinstance(func, relax.Function):
                continue
            updated_func = self.visit_expr(func)
            self.builder_.update_func(global_var, updated_func)

        return self.builder_.get()


def map_matmul(bb, call):
    x, w = call.args
    return bb.call_te(topi.nn.matmul, x, w)

def map_add(bb, call):
    a, b = call.args
    return bb.call_te(topi.add, a, b)

def map_relu(bb, call):
    return bb.call_te(topi.nn.relu, call.args[0])

def map_transpose(bb, call):
    return bb.call_te(topi.transpose, call.args[0], )

op_map = {
  "relax.matmul": map_matmul,
  "relax.add": map_add,
  "relax.nn.relu": map_relu,
  "relax.permute_dims": map_transpose
}

@tvm.ir.transform.module_pass(opt_level=0, name="LowerToTensorIR")
class LowerToTensorIRPass:
    """The wrapper for the LowerTensorIR pass."""
    def transform_module(self, mod, ctx):
        return LowerToTensorIR(mod, op_map).transform()


MLPModelTIR = LowerToTensorIRPass()(MLPFused)
MLPModelTIR.show()
```

Note that in the above code. `fused_matmul_add0` and `fused_matmul_add1` still are high-level relax functions that calls into the corresponding TensorIR matmul and add functions. We can turn them into a single TensorIR function, which then can be used for follow-up optimization and code generation phases.

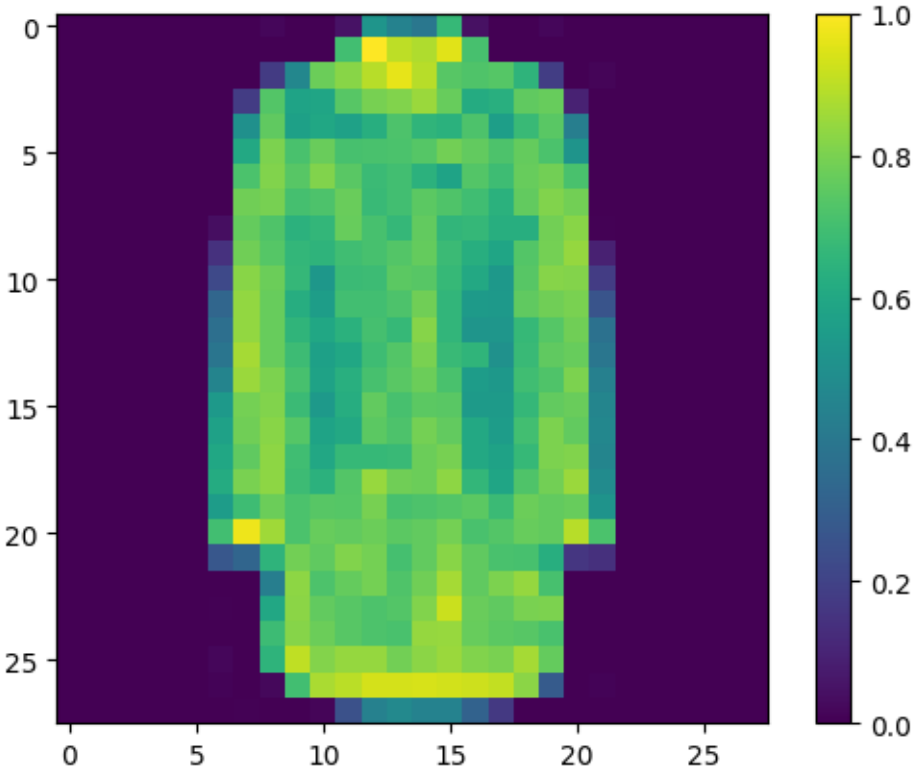
```
MLPModelFinal = relax.transform.FuseTIR()(MLPModelTIR)
MLPModelFinal.show()
```

7.6 Build and Run

We can go ahead and build the final module and try it out on an example picture.

```
# Hide outputs
import torch
import torchvision

test_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=1,␣
↪shuffle=True)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

img, label = next(iter(test_loader))
img = img.reshape(1, 28, 28).numpy()
```

```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(img[0])
plt.colorbar()
plt.grid(False)
plt.show()

print("Class:", class_names[label[0]])
```




```
Class: Dress
```

```
ex = relax.build(MLPModelFinal, target="llvm")
vm = relax.VirtualMachine(ex, tvm.cpu())
data_nd = tvm.nd.array(img.reshape(1, 784))

nd_res = vm["main"](data_nd)

pred_kind = np.argmax(nd_res.numpy(), axis=1)
print("MLPModule Prediction:", class_names[pred_kind[0]])
```

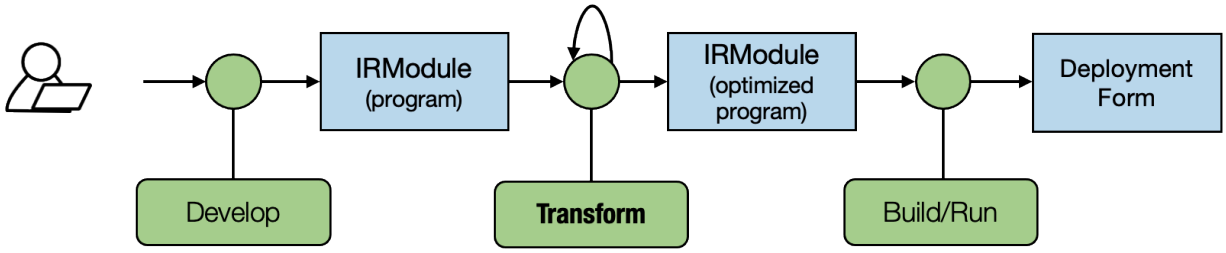
```
MLPModule Prediction: Coat
```

7.7 Discussion

This section comes back to our common theme of **transformation** among computational graphs. Despite being minimum, this sequence of transformations covers two important optimizations we commonly do in MLC process – fusion and loop level code lowering.

Real-world MLC process can contain more powerful and robust transformations. For example, our fusion pass can create duplicated dense computations in which a dense operator is referenced in two follow-ups add operations. A robust fusion pass will detect that and choose to skip such cases. Additionally, we do not want to have to write down rules for each combination. Instead, TVM's internal fusor will analyze the TensorIR function loop patterns and use them in fusion decisions.

Notably, each of these transformations is composable with each other. For example, we can choose to use our version of customized fusor to support additional new fusion patterns that we want to explore and then feed into an existing fusor to handle the rest of the steps.



7.8 Summary

- We can optimize tensor programs by rewriting computational graph data structures.
- Visitor pattern to rewrite call nodes.
- We can perform computational graph transformations, such as fusion and loop-level program lowering.