Machine Learning Compilation

Tensor Program Abstraction

Tianqi Chen

MACHINE LEARNING COMPILATION

OctoML

Outline

Primitive Tensor Function

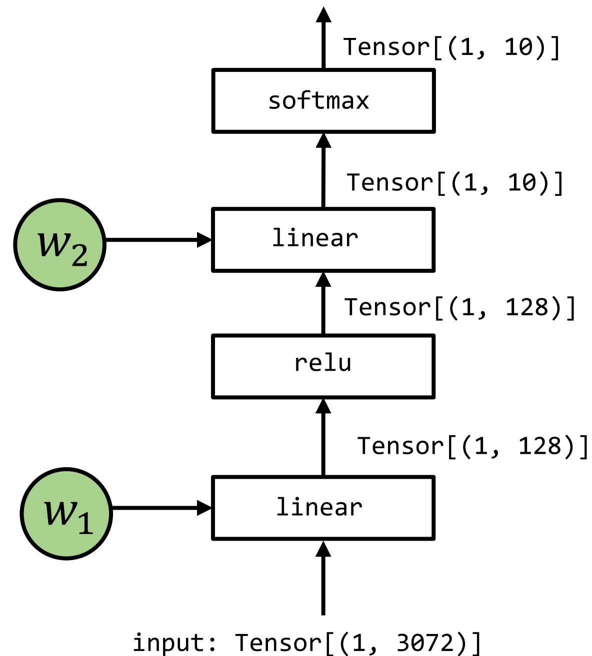
Tensor Program Abstraction

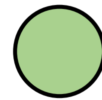
Outline

Primitive Tensor Function

Tensor Program Abstraction

Recap: Key Elements in Machine Learning Compilation

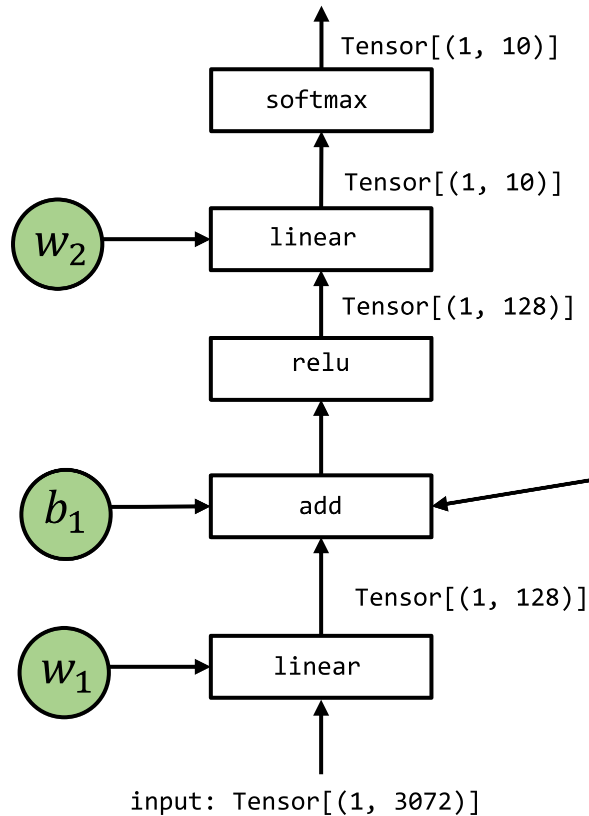




Tensor multi-dimensional array that stores the input, output and intermediate results of model executions.

Tensor Functions that encodes computations among the input/output. Note that a tensor function can contain multiple operations

Primitive Tensor Function



Primitive Tensor Function a tensor function that corresponds to a single "unit" of computational operation.

An example instance of primitive tensor function.

Note: what computations are "unit" can change as we transform the end to end tensor function (e.g. fusing two unit operator functions together)

Primitive Tensor Functions in ML Frameworks

```
import torch

c = torch.empty((128,), dtype=torch.float32)
a = torch.tensor(np.arange(128, dtype="float32"))
b = torch.tensor(np.ones(128, dtype="float32"))

torch.add(a, b, out=c)
```

Note: we explicit allocate output memory in this example so underlying tensor primitive function implementation do not need to handle memory allocation and type conversion. The actual torch.add implementation can be more complicated and goes beyond this particular application scenario.

Abstractions for Primitive Tensor Function

Abstraction refers to different ways to represent the same system interface.

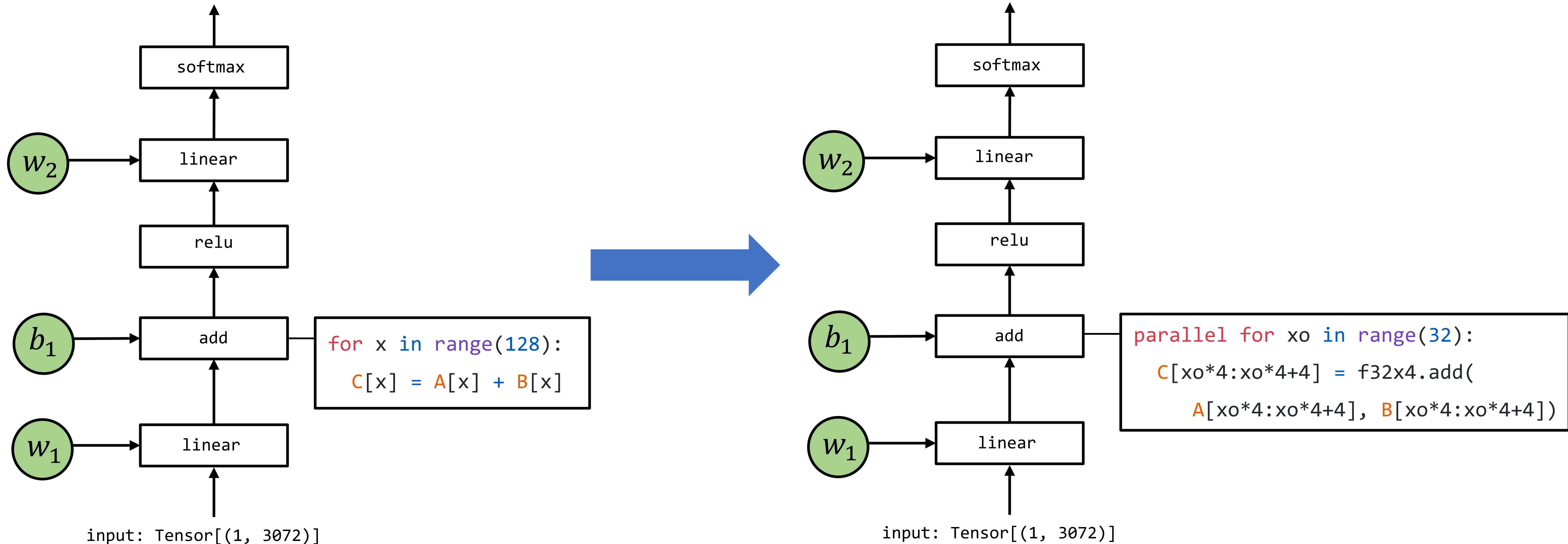
```
torch.add
```

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

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

Three ways to represent the same tensor operator function that adds two vectors of length 128. The python and c version can be viewed as possible **implementations** of higher-level abstraction (torch.add).

MLC via Primitive Function Transformation



One most common MLC process that many frameworks offer is to transform the implementations of primitive functions(or dispatch them in runtime) to more optimized ones based on the environment.

Primitive Function Transformation

Given the overall execution structure remain the same, we only need to focus on transforming the primitive function itself.

```
for x in range(128):
  C[x] = A[x] + B[x]
```

```
parallel for xo in range(32):
  C[xo*4:xo*4+4] = f32x4.add(
     A[xo*4:xo*4+4], B[xo*4:xo*4+4])
```

Approaches for primitive function transformation

Remap to library calls: e.g. cuda, add => cudaAdd (will discuss in incoming lectures)

Fine grained program transformation

Different approaches may require different kinds of abstractions.

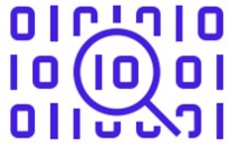
Outline

Primitive Tensor Function

Tensor Program Abstraction

Tensor Program Abstraction

Tensor Programs



Tensor program abstractions focus on loop and layout transformation for fused operators.

```
from tvm.script import tir as T

@T.prim_func
def main(A: T.Buffer[128, "float32"],
         B: T.Buffer[128, "float32"],
         C: T.Buffer[128, "float32"]):
    for i in range(128):
        with T.block("C"):
            vi = T.axis.spatial(128, i)
            C[vi] = A[vi] + B[vi]
```

An example tensor program instance

Key Elements of a Tensor Program

```
from tvm.script import tir as T

@T.prim_func
def main(A: T.Buffer[128, "float32"],
         B: T.Buffer[128, "float32"],
         C: T.Buffer[128, "float32"]):
    for i in range(128):
        with T.block("C"):
            vi = T.axis.spatial(128, i)
            C[vi] = A[vi] + B[vi]
```

(Multi-dimensional) buffers that holds the input, output, and intermediate results.

Loop nests that drive compute iterations.

Computations statement.

A typical tensor program abstraction contains multi-dimensional buffers, loop nests that drive compute iterations and finally computation statement itself.

Why do we need Tensor Program Abstraction

Carefully designed tensor program abstraction enables program-based transformations among variants without reimplementation from scratch

Initial state

```
for x in range(128):
  C[x] = A[x] + B[x]
```

Program-based transformations

```
xo, xi = split(x, 4)
```

```
parallelize(xo)
```

```
vectorize(xi)
```

Transformed program

```
parallel for xo in range(32):
  C[xo*4:xo*4+4] = f32x4.add(
     A[xo*4:xo*4+4], B[xo*4:xo*4+4])
```

Enables specialization (to specific shape or device)

Example Transformation: Loop Splitting

Code

```
for x in range(128):
  C[x] = A[x] + B[x]
```

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

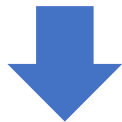
```
x = get_loop("x")
xo, xi = split(x, 4)
```

Pseudo code to for demonstrating ideas only

Example Transforming Loops: Loop Reorder

Code

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```



```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
```

Pseudo code to for demonstrating ideas only

Example Transformation: Thread Binding

Code

```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
def gpu_kernel():
  C[threadId.x * 4 + blockIdx.x] = . . .
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
bind_thread(xo, "threadIdx.x")
bind_thread(xi, "blockIdx.x")
```

Pseudo code to for demonstrating ideas only

We cannot Arbitrarily Transform any Program

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + C[max(xo * 4 + xi - 1, 0)]
```

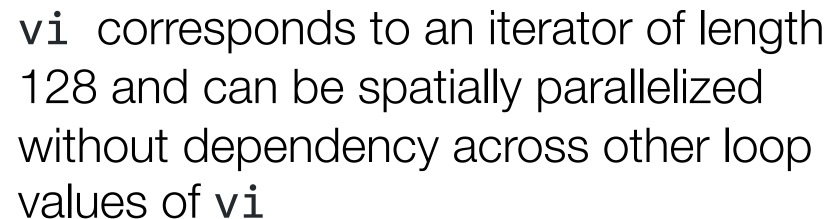
Extra structure information (or analysis to recover the structure information) is needed in actual tensor program abstractions to avoid such kinds of transformations.

Extra Structure in Tensor Program Abstraction

```python
from tvm.script import tir as T

@T.prim_func
def main(A: T.Buffer[128, "float32"],
         B: T.Buffer[128, "float32"],
         C: T.Buffer[128, "float32"]):
    for i in range(128):
        with T.block("C"):
            vi = T.axis.spatial(128, i)
            C[vi] = A[vi] + B[vi]
```

Extra information about iteration



The additional structure information (about iterators) helps us to detect incorrect transformations and provide more information for the MLC process. We can usually obtain these information from the definition of primitive tensor functions.

Tensor Program Transformation in Action

Summary

- Primitive tensor function refers to the single unit of computation in model execution.
  - One important MLC process is 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.