ECE 449/590 – OOP and Machine Learning
Lecture 03 EasyNN

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

August 29, 2022

Outline

Reading Assignment

- This lecture: Project Introduction
- Next lecture: Accelerated C++ 0–2

Outline

Data Flow Graphs

EasyNN

Specification vs. Implementation

- Any non-trivial functionality must involve some kind of computation.
  - E.g. for machine and deep learning.
- Two aspects of computations
  - Specification: what the computation is supposed to do, e.g. to add two matrices.
  - Implementation: how to implement the computation, e.g. using NumPy.
  - Most of time, both aspects are addressed together when we write a program.
- It is beneficial to separate the specification and the implementation.
  - Specification tends to be more intuitive.
  - Implementation tends to have a lot of details to achieve performance goals.
  - Tools may exist to automate the process to generate alternative implementations, e.g. to implement a machine and deep learning algorithm on different libraries and platforms like NumPy, GPU, and FPGA.

Data Flow Graphs (DFG)

- A model to specify computations.
  - A graph consisting of nodes and (directed) arcs.
  - Nodes represent operations.
  - Arcs indicate inputs and outputs of the operations.
  - No cycle: it is a directed acyclic graph (DAG) where the computation can be performed by executing each node following their topological order.
- A set of statements without branches can be transformed into a DFG.
  - Intermediate variables may be eliminated.
  - Ordering of operations may be relaxed.
  - DFG is an intermediate representation (IR) of a program.

Example Statements

```
inputs/outputs: a, b, c, d, f

  f = (a+b)*(c-d);
```

- In this example, the operations are `*`, `+`, and `-`.
  - Any operaions matching the computation itself can be used.

Example Statements (Cont.)

```
inputs/outputs: a, b, c, d, f
temporary variables: t0, t1

  t0 = a+b;
  t1 = c-d
  f = t0*t1;
```

- It is usually more convenient to represent a DFG in the static single assignment (SSA) form.
  - Break complex expressions down into expressions with single operations.
  - Introduce temporary variables so that each variable is assigned only once.

Capturing DFG in SSA Form

- A compiler frontend: a special program that is able to extract DFG from another program in the SSA form.
  - As the specification of the computation.
  - So that implementations may be generated at a later time.
- Not trivial: consider the expression $a + b * c$
  - The compiler frontend needs to process the expression letter by letter.
  - The meaning of $a + b$ is not clear since the expression could be $a + b + c$ or $a + b * c$.
- Can we leverage a programming language where there is already a compiler frontend?
  - So that when we write $a + b * c$, we would like to capture the two operations and their inputs/outputs, instead of actually computing it.
  - Possible with most languages and most convenient with a very expressive language like Python and C++.

Outline

Data Flow Graphs

EasyNN

EasyNN

- A Python library we will develop for the course projects.
  - Allow to specify a complex computation, e.g. a deep learning algorithm, intuitively in Python.
  - Define and capture the DFG of the computation in the SSA form.
- A reference implementation based on NumPy is provided.
- You are exprected to learn how the library works, to provide a C++ based implementation, and to extend it for the course projects.

Inputs

```
>>> import easynn as nn
>>> a = nn.Input("a")
>>> print(a.statements())
t0 = a.Input()
```

- EasyNN operators are defined in `easynn.py` and you may refer to them from a different Python program by `import` it.
  - Rename the library as `nn` for convenience.
- Always start by specifying the inputs of the computation via `Input`.
  - We'll need to specify the name of the inputs, e.g. `"a"`.
- Use `statements()` to display the DFG in the SSA form.

Operators

```
>>> b = a+a
>>> print(b.statements())
t0 = a.Input()
t1 = .Add(t0,t0)
```

- A few operators like `*`, `+`, and `-` are supported.
- For simplicity, all EasyNN operators have a single output.
- Note that `b = a+a` does not perform any computation but specifies an operation to add two inputs.

Intermediate Variables

```
>>> c = a+b*b
>>> print(c.statements())
t0 = a.Input()
t1 = .Add(t0,t0)
t2 = .Mul(t1,t1)
t3 = .Add(t0,t2)
```

- Intermediat variables are introduced to brake down the computation into SSA form.
  - Python variable names like `a`, `b`, and `c` are removed from the DFG.

Compiling and Execution

```
>>> import easynn_golden as golden
>>> cc = c.compile(golden.Builder())
>>> cc(a = 1)
5.0
>>> cc(a = 3)
39.0
```

- The reference implementation is defined in `easynn_golden.py`.
- Use `compile()` with the golden `Builder` to compile the captured DFG into a program object, e.g. `cc`.
- Run `cc` by simply calling it with the inputs.

Polymorphism

```
>>> import numpy as np
>>> cc(a = np.array([[0,1],[1,0]]))
array([[4., 1.],
       [1., 4.]])
```

- Inputs as NumPy matrices are also supported.

More Complex Operators

```
>>> relu = nn.ReLU()
>>> d = relu(a)
>>> dd = d.compile(golden.Builder())
>>> dd(a = 1)
1.0
>>> dd(a = -1)
-0.0
>>> dd(a = np.array([[-1,1],[1,-1]]))
array([[-0.,  1.],
       [ 1., -0.]])
```

- More complex operators like `ReLU` are supported.
  - $ReLU(x)$ returns x if $x \geq 0$, or 0 if $x < 0$.
  - Unlike `*`, `+`, and `-`, you'll need to create the operator before you could use them, as many of them requires additional parameters.

Constants

```
>>> e = nn.Const(5)
>>> f = a+e
>>> ff = f.compile(golden.Builder())
>>> ff(a = 1)
6.0
```

- You may use `Const` to introduce constants to the computation.
- Work with Project 1 to learn more about EasyNN.

Summary

- Computation may be specified as DFG, making it possible to generate alternative implementations.