Lesson 19

Week 5

Lesson 17 - STARK implementation
Lesson 18 - Plonk part 1 / Linea
Lesson 19 - Plonk part 2 / Boojum
Lesson 20 - ZKML

Today's topics

- Using the Plonky2 library
- Polynomial permutation
- Hyperplonk
- Plonk implementation in python

Plonky2 Library

See tutorial

Creating a circuit

The prover's claim is

I know x s.t

$x^2 - 4x + 7 =$ some public value

```
// The arithmetic circuit.
let x = builder.add_virtual_target();
let a = builder.mul(x, x);
let b =
builder.mul_const(F::from_canonical_u32(4),
x);
let c = builder.mul_const(F::NEG_ONE, b);
let d = builder.add(a, c);
let e = builder.add_const(d,
F::from_canonical_u32(7));
```

The complete code

```


use anyhow::Result;
use plonky2::field::types::Field;
use plonky2::iop::witness::{PartialWitness,
Witness};
use
```

```rust
plonky2::plonk::circuit_builder::CircuitBuild
er;
use
plonky2::plonk::circuit_data::CircuitConfig;
use plonky2::plonk::config::{GenericConfig,
PoseidonGoldilocksConfig};

/// An example of using Plonky2 to prove a
statement of the form
/// "I know x² - 4x + 7".
fn main() -> Result<()> {
const D: usize = 2;
type C = PoseidonGoldilocksConfig;
type F = <C as GenericConfig<D>>::F;

let config =
CircuitConfig::standard_recursion_config();
let mut builder = CircuitBuilder::<F,
D>::new(config);

// The arithmetic circuit.
let x = builder.add_virtual_target();
let a = builder.mul(x, x);
let b =
builder.mul_const(F::from_canonical_u32(4),
x);
let c = builder.mul_const(F::NEG_ONE, b);
let d = builder.add(a, c);
let e = builder.add_const(d,
F::from_canonical_u32(7));
```

```
// Public inputs are the initial value
(provided below) and the result (which is
generated).
builder.register_public_input(x);
builder.register_public_input(e);
let mut pw = PartialWitness::new();
pw.set_target(x, F::from_canonical_u32(1));
let data = builder.build::<C>();
let proof = data.prove(pw)?;
println!(
"x² - 4 *x + 7 where x = {} is {}",
proof.public_inputs[0],
proof.public_inputs[1]
);
data.verify(proof)
}
```

Hyperplonk

From the hyperplonk paper

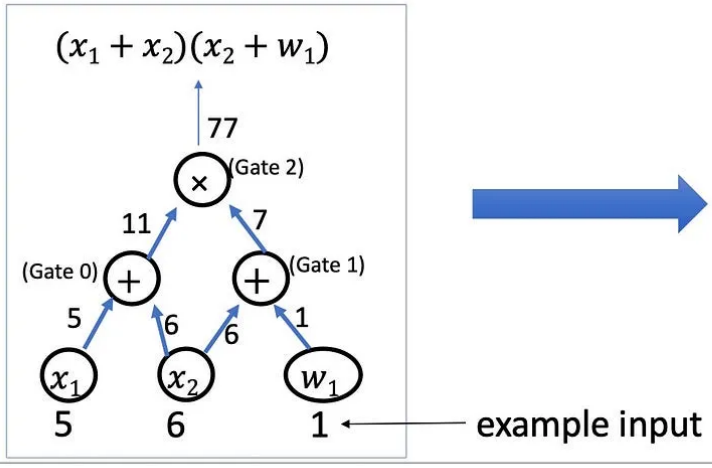
"Plonk is quite flexible: it supports circuits with low-degree “custom” gates as well as circuits with lookup gates (a lookup gate ensures that its input is contained in a predefined table). For large circuits, the bottleneck in generating a Plonk proof is the need for computing a large FFT.

HyperPlonk is an adaptation of Plonk to the boolean hypercube, using multilinear polynomial commitments. HyperPlonk retains the flexibility of Plonk but provides several additional benefits. First, it avoids the need for an FFT during proof generation. Second, and more importantly, it supports custom gates of much higher degree than Plonk without harming the running time of the prover. Both of these can dramatically speed up the prover’s running time."

From article

PLONK: a poly-IOP for a general circuit $C(x, w)$ [GWC19]

Step 1: compile circuit to a computation trace (gate fan-in = 2)



The computation trace:

inputs:	5,	6,	1
Claimed output: 77			
Gate 0:	5,	6,	11
Gate 1:	6,	1,	7
Gate 2:	11,	7,	77

left inputs | right inputs | outputs

Encoding the trace as a polynomial

In our example:

inputs:	$P(\omega^{-1}) = 5,$	$P(\omega^{-2}) = 6,$	$P(\omega^{-3}) = 1,$
gate 0:	$P(\omega^{0}) = 5,$	$P(\omega^{1}) = 6,$	$P(\omega^{2}) = 11,$
gate 1:	$P(\omega^{3}) = 6,$	$P(\omega^{4}) = 1,$	$P(\omega^{5}) = 7,$
gate 2:	$P(\omega^{6}) = 11,$	$P(\omega^{7}) = 7,$	$P(\omega^{8}) = 77$

$$P(X) = 5 + 4x + 9x^3 + 13x^4 + 15x^5 + 4x^6 + 4x^7 + 6x^8 + 2x^9 + 13x^{10} + 15x^{11} \in \mathbb{F}_{17}$$

degree(P) = 11

inputs:	5,	6,	1
Gate 0:	5,	6,	11
Gate 1:	6,	1,	7
Gate 2:	11,	7,	77

The runtime of an FFT is proportional to $n \log_2(n)$ for a circuit of size n.

When n is small or medium-sized, this does not present a challenge but for ns larger than about a million (and $\log_2(n)$ about 20) FFTs become a significant factor. This is, especially an issue for systems like zk-

rollups and zkEVMs when circuits can have almost a billion gates.

The goal of Hyperplonk is to remove the requirements for FFTs from Plonk to make it more scalable and suitable for custom hardware.

Instead of encoding a polynomial $P(X)$ with one variable, we use multiple variables ($\log_2(n)$ to be precise) and encode a polynomial $P(X_1, X_2, ..., X_\mu)$. It turns out that with $\log_2(n)$ variables we can find a polynomial that encodes n values but is at most linear in every variable.

More precisely, we can easily in time n encode a so-called multi-linear polynomial that has a certain set of values at the values $(0,0,\ldots 0)$ to $(1,1,\ldots,1)$.

This is called the boolean hypercube, as each variable takes on the value either 0 or 1 and as you may have guessed, this hypercube inspired the name for HyperPlonk.

Implementing Plonk in Python

See this repo