Efficient prime-field arithmetic in Rust

`Simon Shine <simon@neptune.cash>`

Slides available on:

https://github.com/sshine/prime-field-benchmarks/

Who am I?

I'm Simon.

I work at Triton Software 🇨🇭, and I make zero-knowledge cryptography for a privacy-focused blockchain called Neptune: https://neptune.cash/

Who am I?

I'm Simon.

In my spare time, I study Chinese and practice time travel (mostly forwards, in kayaks).

Disclaimer

I'm not a mathematician. Don't hesitate to add to what I say. Also, I didn't invent anything here. All of this is re-discovered. I try to give credit.

tl;dr

I $+$ and $\times$ mod p faster using a combination of hacks, tricks, and knowledge of p.

Field arithmetic

A **field** is a set S with $+$ and $\times$ where

- $+$ and $\times$ associate, commute and distribute
- $\exists 0 \in S$ so for all $a \in S$, $0 + a = a$ and $a + 0 = a$.
- $\exists 1 \in S$ so for all $a \in S$, $1 \times a = a$ and $a \times 1 = a$.
- $\forall a \in S$, $\exists b \in S$ so $a + b = b + a = 0$
- $\forall a \in S$, $a \neq 0$, $\exists b \in S$ so $a \times b = b \times a = 1$
- $0 \neq 1$

Field arithmetic

It just means you can $+, -, \times, \div$ as you'd expect.

Reals and rationals are examples of infinite fields.

Most people call this "**math**".

Finite-field arithmetic

An example of an (efficient) **finite field** is `u64` with overflow (aka $\mathrm{GF}(2^{64})$).

Most programmers call this "**math**".

Prime-field arithmetic

A **prime field** $\mathbb{F}_p$ is a finite field that overflows at p.
Or said with Rust code,

```rust
use std::ops::{Add, Mul, Sub, Div};
use num_traits::{Zero, One};

pub trait PrimeField:
    Zero + One + Add + Mul + ModReduce + Sub + Div + Eq {}

pub trait ModReduce {
    #[must_use]
    fn mod_reduce(product: u128) -> u64;
}
```

Inefficient prime-field arithmetic in Rust

```rust
// 2^64 - 2^32 + 1
pub const P64: u64 = 0xffff_ffff_0000_0001;
pub const P128: u128 = 0xffff_ffff_0000_0001;

pub fn add(x: u64, y: u64) -> u64 {
    let sum: u128 = x as u128 + y as u128;
    (sum % P128) as u64
}

pub fn mul(x: u64, y: u64) -> u64 {
    let product: u128 = x as u128 * y as u128;
    (product % P128) as u64
}
```

Efficient prime-field $+$

```rust
pub fn add_fast(x: u64, y: u64) -> u64 {
    let mut sum: u128 = x as u128 + y as u128;
    if sum > P128 {
        sum -= P128;
    }
    sum as u64
}
```

Efficient prime-field $\times$

(for $p = 2^{64} - 2^{32} + 1$*)*

Credit: cp4space.hatsya.com's blog post:
An efficient prime for number-theoretic transforms

- Elements in $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$ fit nicely inside a 64-bit machine word, and "mod p" is possible without multiplication or division.

Efficient prime-field $\times$

(for $p = 2^{64} - 2^{32} + 1$)

Any non-negative integer less than 2^{159} can be written as $A2^{96} + B2^{64} + C$ where A is a 63-bit integer, B is a 32-bit integer, and C is a 64-bit integer.

Efficient prime-field $\times$

(for $p = 2^{64} - 2^{32} + 1$*)*

Since 2^{96} is congruent to -1 modulo p, this can be rewritten as $B2^{64} + (C - A)$. If $A > C$, $B2^{64}$ could underflow, in which case we can add p, resulting in a 96-bit integer.

Efficient prime-field $\times$

(for $p = 2^{64} - 2^{32} + 1$*)*

To reduce this to a 64-bit integer, 2^{64} is congruent to $2^{32} - 1$, so we can multiply B by 2^{32} using a binary shift and a subtraction, and then add it to the result. We might encounter an overflow, but we can correct for that by subtracting p.

Efficient prime-field $\times$

(*for* $p = 2^{64} - 2^{32} + 1$)

See the source code. How fast is it, then?

`cargo criterion`

11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz

```
add/baseline/1000        time:   [0.0000 ps 0.0000 ps 0.0000 ps]
add/mod/1000             time:   [2.5268 µs 2.5290 µs 2.5317 µs]
add/fast/1000            time:   [1.0189 µs 1.0195 µs 1.0202 µs]
add/winterfell/1000      time:   [1.0564 µs 1.0573 µs 1.0581 µs]

mul/baseline/1000        time:   [0.0000 ps 0.0000 ps 0.0000 ps]
mul/mod/1000             time:   [2.5743 µs 2.5756 µs 2.5770 µs]
mul/reduce159/1000       time:   [1.6030 µs 1.6041 µs 1.6052 µs]
mul/reduce_montgomery/1000
                         time:   [1.3197 µs 1.3206 µs 1.3215 µs]
```

`cargo criterion`

MacBook Pro M1 2021

```
add/mod/1000                time:   [7.2183 µs 7.2243 µs 7.2303 µs]
add/fast/1000               time:   [1.3538 µs 1.3570 µs 1.3602 µs]
add/winterfell/1000         time:   [1.2872 µs 1.2876 µs 1.2880 µs]
mul/mod/1000                time:   [13.447 µs 13.467 µs 13.487 µs]
mul/reduce159/1000          time:   [1.0562 µs 1.0569 µs 1.0576 µs]
mul/reduce_montgomery/1000
                            time:   [975.32 ns 975.60 ns 975.94 ns]
```

Godbolt!

See the disassembly for x86_64 and aarch64.

That was $+$ and $\times$... what else?

For prime fields and uni-/multivariate polynomials:

- `AddAssign` (`+=`), `MulAssign` (`*=`), when possible
- `.mod_pow()` with halving strategy ($x^{2n} = x^n \cdot x^n$)
- Generally, don't divide, multiply by inverses.
- Specialize `.square()` .
- Only re-allocate coefficients when `rhs.len() > lhs.len()` .
- Short-circuit polynomial multiplication when operand is $f(x) = 0$ or $f(x) = 1$.

EOF

See Alan Szepieniec's STARK anatomy tutorial:

https://neptune.cash/learn/stark-anatomy/