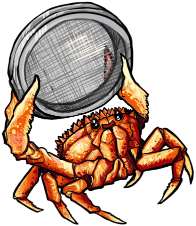
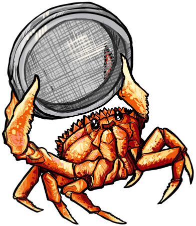
 

Going as fast as possible in Rust: An in-depth look at optimizing the Quadratic Sieve

Christopher Swenson
swenson.io

SeaGL 2023; November 4, 2023

Who am I?

Christopher Swenson, Ph.D.

Currently at HashiCorp (prev. Google, Twilio, Simple, US Government).

I love programming languages, math, and cryptography.

I have a book, *Modern Cryptanalysis*.

Outline

1. Quadratic Sieve explanation
2. Basic Rust implementation
3. Double Optimization with Doctor Skelebone

Factoring

The goal of factoring is to factor a number N, that is, find the prime numbers that divide N.

Cryptography uses the difficulty of factoring large numbers as a basis of security, e.g., RSA, and hence SSL/TLS.

N is typically thousands of bits large.

Quadratic Sieve

The Quadratic Sieve is an algorithm for factoring a number that works well for numbers that are ≈ 100–300 bits in size.

The Quadratic Sieve is not the fastest algorithm, but it is relatively simple, and for the most part, it only requires basic algebra to understand.

It's also a great programming meditation that everyone should do.

msieve

`msieve`:

- msieve is a program written primarily in in 2004–2009 primarily by Jason Papadopoulos.
- It contains several factoring algorithm implementations, notably the Quadratic Sieve and the General Number Field Sieve.
- Highly optimized C, some assembly.
- Meant for x86.

...what if I reimplemented it in 2023 in Rust? How hard could it be?

(single thread, single core, all Rust, no GPU)

Succulents

ReductRs
@reduct_rs

'I Think I Can Take On A New Coding Side Project,' Says Woman Who Can't Keep Succulents Alive

ReductRs
@reduct_rs

Developer Has Own Definition of "It'll Take a Week"

Difference of Squares

$$a^2 - b^2$$

Difference of Squares

$$a^2 - b^2$$

$$(a + b) \cdot (a - b)$$

Fermat's Method

1. Search for perfect squares around $\sqrt{N}$.
2. If you find one, say, a^2, check if $a^2 - N$ is also square, b^2.
3. If true, then the factors of N $(= a^2 - b^2)$ will generally be evenly distributed between $a + b$ and $a - b$.
4. Can compute $\gcd(a + b, N)$ and $\gcd(a - b, N)$ to try to find the factors.

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.
- Do this by looking for $a_i^2 - N$ entries that have common factors.

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.
- Do this by looking for $a_i^2 - N$ entries that have common factors.
- (Each a_i where $a_i^2 - N$ is *smooth* is called a *relation*.)

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.
- Do this by looking for $a_i^2 - N$ entries that have common factors.
- (Each a_i where $a_i^2 - N$ is *smooth* is called a *relation*.)
- This will give us $a_0^2 \cdot a_1^2 \cdots \equiv b^2 \pmod{N}$,

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.
- Do this by looking for $a_i^2 - N$ entries that have common factors.
- (Each a_i where $a_i^2 - N$ is *smooth* is called a *relation*.)
- This will give us $a_0^2 \cdot a_1^2 \cdots \equiv b^2 \pmod{N}$,
- hence $a_0^2 \cdot a_1^2 \cdots - b^2 = k \cdot N$ for some k.

Quadratic Sieve

- The Quadrative Sieve (QS) is an refinement of Fermat's Method.
- Instead of looking for single a^2 where $a^2 - N$ is a perfect square,
- find multiple a_0^2, a_1^2, ..., where $(a_0^2 - N) \cdot (a_1^2 - N) \cdots$ multiplied together make a perfect square.
- Do this by looking for $a_i^2 - N$ entries that have common factors.
- (Each a_i where $a_i^2 - N$ is *smooth* is called a *relation*.)
- This will give us $a_0^2 \cdot a_1^2 \cdots \equiv b^2 \pmod{N}$,
- hence $a_0^2 \cdot a_1^2 \cdots - b^2 = k \cdot N$ for some k.
- Can try to factor with $\gcd(a_0 \cdot a_1 \cdots \pm b, N)$.

QS tiny example

Example: Let's factor $N = 493$.

$$\left\lceil \sqrt{493} \right\rceil = 23$$

Looking around 23, we find two relations:

QS tiny example

Example: Let's factor $N = 493$.

$$\left\lceil \sqrt{493} \right\rceil = 23$$

Looking around 23, we find two relations:

$$26^2 - 493 = 676 - 493 = 183 = \underline{3 \cdot 61}$$

QS tiny example

Example: Let's factor $N = 493$.

$$\left\lceil \sqrt{493} \right\rceil = 23$$

Looking around 23, we find two relations:

$$26^2 - 493 = 676 - 493 = 183 = \underline{3 \cdot 61}$$

$$35^2 - 493 = 1225 - 493 = 732 = \underline{2^2 \cdot 3 \cdot 61}$$

QS tiny example

Example: Let's factor $N = 493$.

$$\left\lceil \sqrt{493} \right\rceil = 23$$

Looking around 23, we find two relations:

$$26^2 - 493 = 676 - 493 = 183 = \underline{3 \cdot 61}$$

$$35^2 - 493 = 1225 - 493 = 732 = \underline{2^2 \cdot 3 \cdot 61}$$

So we can construct:

$$26^2 \cdot 35^2 - 2^2 \cdot 3^2 \cdot 61^2 = k \cdot 493$$

QS tiny example

$$26^2 \cdot 35^2 - 2^2 \cdot 3^2 \cdot 61^2 = k \cdot 493$$

QS tiny example

$$26^2 \cdot 35^2 - 2^2 \cdot 3^2 \cdot 61^2 = k \cdot 493$$

$$26 \cdot 35 + 2 \cdot 3 \cdot 61, \quad 26 \cdot 35 - 2 \cdot 3 \cdot 61$$

QS tiny example

$$26^2 \cdot 35^2 - 2^2 \cdot 3^2 \cdot 61^2 = k \cdot 493$$

$$26 \cdot 35 + 2 \cdot 3 \cdot 61, \quad 26 \cdot 35 - 2 \cdot 3 \cdot 61$$

$$1276, \quad 544$$

QS tiny example

$$26^2 \cdot 35^2 - 2^2 \cdot 3^2 \cdot 61^2 = k \cdot 493$$

$$26 \cdot 35 + 2 \cdot 3 \cdot 61, \quad 26 \cdot 35 - 2 \cdot 3 \cdot 61$$

$$1276, \quad 544$$

$$\mathsf{gcd}(1276, 493) = \underline{29}$$

$$\mathsf{gcd}(544, 493) = \underline{17}$$

$$17 \cdot 29 = 493$$

More QS explanation

- For larger N, need more than 2 relations, and they'll span many prime factors.
- For the product the RHS's of the relations to be a square, need to track if the exponents are even.
- Can do that in a binary matrix.
- Limit our relations to only those whose RHS contains primes smaller than a bound, called the small-prime bound.
- Can *sieve* for these relations to speed up the search.

Sieve of Eratosthenes review

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

Sieve of Eratosthenes review

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

Sieve of Eratosthenes review

0	1	2	3	~~4~~	5	~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19
~~20~~	21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29
~~30~~	31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39

Sieve of Eratosthenes review

0	1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19
~~20~~	~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29
~~30~~	31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~

Sieve of Eratosthenes review

0	1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19
~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29
~~30~~	31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~

Sieve of Eratosthenes review

0	1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19
~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29
~~30~~	31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~

Counting factors

0	1	2	3	2 4	5	2, 3 6	7	2 8	3 9
2, 5 10	11	2, 3 12	13	2, 7 14	3, 5 15	2 16	17	2, 3 18	19
2, 5 20	3, 7 21	2, 11 22	23	2, 3 24	5 25	2, 13 26	3 27	2, 7 28	29
2, 3, 5 30	31	2 32	3, 11 33	2, 17 34	5, 7 35	2, 3 36	37	2, 19 38	3, 13 39

Counting factors

0	1	2	3	1.0 4	5	2.6 6	7	1.0 8	1.6 9
3.3 10	11	2.6 12	13	3.8 14	3.9 15	1.0 16	17	1.6 18	19
3.3 20	4.4 21	4.5 22	23	2.6 24	2.3 25	4.7 26	1.6 27	3.8 28	29
4.9 30	31	1.0 32	5.0 33	5.1 34	5.1 35	2.6 36	37	5.2 38	5.3 39

QS Sieve

We can do a similar trick to sieve for points x where $x^2 - N$ is divisible by primes 2, 3, ...

Start at $x = \left\lceil \sqrt{N} \right\rceil$, and note the points where $x^2 - N$ are divisible by p.

QS Sieve

We can do a similar trick to sieve for points x where $x^2 - N$ is divisible by primes 2, 3, ...

Start at $x = \left\lceil \sqrt{N} \right\rceil$, and note the points where $x^2 - N$ are divisible by p.

Example: $N = 493$, $\lceil \sqrt{493} \rceil = 23$.

$p = 2$: $x = 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49$
$p = 3$: $x = 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 38, 40, 41, 43, 44, 46, 47, 49$
$p = 11$: $x = 25, 30, 36, 41, 47$
$p = 13$: $x = 31, 34, 44, 47$

QS Sieve

We can do a similar trick to sieve for points x where $x^2 - N$ is divisible by primes 2, 3, ...

Start at $x = \left\lceil \sqrt{N} \right\rceil$, and note the points where $x^2 - N$ are divisible by p.

Example: $N = 493$, $\lceil \sqrt{493} \rceil = 23$.

$p = 3$: $x = \underline{23}, 25, \underline{26}, 28, \underline{29}, 31, \underline{32}, 34, \underline{35}, 37, \underline{38}, 40, \underline{41}, 43, \underline{44}, 46, \underline{47}, 49$
$p = 11$: $x = \underline{25}, 30, \underline{36}, 41, \underline{47}$
$p = 13$: $x = \underline{31}, 34, \underline{44}, 47$

Basic QS Code

Finally, some code!

```
let mut sieve = vec![0f32; area];
for p in primes.iter() {
  let sqrtp = (sqrt_num.clone() % p).to_i64().unwrap();
  let logp = (*p as f32).log2();
  for r in poly::get_roots(&num, *p).into_iter() {
    let mut idx = (r as i64 - sqrtp).rem_euclid(*p as i64
        ) as usize;
    while idx < area {
        sieve[idx] += logp;
        idx += *p as usize;
    }
  }
}
```

Scanning

After we have sieved out a bunch of primes, we need to scan the array to large numbers.

```
let mut smooth_points = vec![];
let cutoff = (num.to_f64().unwrap().log2() * 3.0 / 5.0).
   to_f32().unwrap();
for i in 0..area {
    if sieve[i] > cutoff {
        smooth_points.push(i)
    }
}
```

Simple enough, for now.

Smoothness Checking

After scanning, we have a list of points, but we need to

1. Check that they are actually smooth
2. Get a list of all factors for the smooth ones

```
for point in smooth_points.iter() {
  let smooth_check = BigInt::from(point + sqrt_num.clone
     ()) * BigInt::from(point + sqrt_num.clone()) - num.
     clone();
  let smooth_factors = small_factor(&smooth_check, &
     primes);
  let check = mul_vec(&smooth_factors);
  if check != smooth_check {
      continue;
  }
  relations.push(Relation {
      num: point + sqrt_num.clone(),
      factors: smooth_factors,
  });
}
```

The Matrix

Take the smooth numbers and their factors and build a matrix.

If we consider our earlier example, with our relations:

1. $26^2 - 493 = 3 \cdot 61$
2. $35^2 - 493 = 2^2 \cdot 3 \cdot 61$

The matrix of exponents would look like:

$$\left[\begin{array}{c|ccc} & 2 & 3 & 61 \\ \hline 26 & 0 & 1 & 1 \\ 35 & 2 & 1 & 1 \end{array}\right]$$

Building the matrix is mostly tedious bookkeeping, so we'll skip the code.

LA (cont'd.)

$$\left[\begin{array}{c|ccc} & 2 & 3 & 61 \\ \hline 26 & 0 & 1 & 1 \\ 35 & 2 & 1 & 1 \end{array}\right]$$

LA (cont'd.)

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

In general, we reduce the entries in the bottom right to binary and find a combination of rows that sum to 0. For now, just write the matrix out to disk in ASCII and use, e.g., Sage, to find the left kernel.

Last Step

The output of the LA is a list of relations that when multiplied together give us a^2 and b^2 such that $a^2 - b^2 = k \cdot N$. We need to use that information to find a and b. This is another matter that is mostly bookkeeping:

Last step (code)

```
let mut h = HashMap::new();
for i in 0..mat.nrows() {
  if result_vector[i] == 0 {
      continue;
  }
  for f in relations[i].factors.iter() {
      h.insert(f, h.get(&f).unwrap_or(&0u32) + 1);
  }
  a = a * (relations[i].num.clone());
}
for (f, e) in h.iter() {
    b = (b * BigInt::from_i64(**f).unwrap()
        .modpow(&BigInt::from_u32(e / 2).unwrap(), &num))
        % num.clone();
}
let g1 = (a.clone() + b.clone()).gcd(&num.clone());
let g2 = (a - b).gcd(&num.clone());
```

Then `g1` or `g2` are either 1, N, or a factor of N.

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a "global" `HashMap`.

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a “global” `HashMap`.

```
HashMap<str, f64>
```

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a “global” `HashMap`.

~~`HashMap<str, f64>`~~
`HashMap<&str, f64>`

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a “global” `HashMap`.

~~`HashMap<str, f64>`~~
~~`HashMap<&str, f64>`~~

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a “global” `HashMap`.

~~`HashMap<str, f64>`~~
~~`HashMap<&str, f64>`~~
`HashMap<&'static str, f64>`

Timing

The first step to optimizing is to collect timing info.

Easiest way I can think of it to have a “global” `HashMap`.

~~`HashMap<str, f64>`~~
~~`HashMap<&str, f64>`~~
~~`HashMap<&'static str, f64>`~~

Not Proud

```
pub struct Timers {
    timers: RefCell<HashMap<String, f64>>,
}
// ...
let timers = Rc::new(Timers::new());
```

Basic timings

Timing for 80-bit number, basic, ext. LA, M2

Parameters: sieve area: $1{,}000{,}000$, fb size: $2{,}000$

Matrix size: 169×194

left kernel	3.473s	99.64%
line sieve	0.003s	0.10%
matrix build	0.001s	0.02%
scan	0.001s	0.01%
smooth check	0.007s	0.20%
sqrt	0.001s	0.03%
total	3.485s	

Cat Ears


ReductRs
@reduct_rs
Coworker Insists Her Rust Code Will Run Faster If She Writes It While Wearing Cat Ears

LA for real

Calling out to an external program to run our LA is a bad idea. Replace with basic, Wikipedia left kernel using `nalgebra::DMatrix<u8>`.

Timing for 80-bit number, basic, own LA, M2

Parameters: sieve area: $1{,}000{,}000$, fb size: $2{,}000$

Matrix size: 150×254

left kernel	0.000s	2.38%
line sieve	0.004s	20.47%
matrix build	0.000s	1.43%
scan	0.000s	2.61%
smooth check	0.009s	50.17%
sqrt	0.004s	22.94%
total	0.017s	

Scale up

Scale up until so we can find the weak points.

Timing for 120-bit number, basic, own LA, M2

Parameters: sieve area: $50{,}000{,}000$, fb size: $28{,}000$
Matrix size: $1{,}466 \times 2{,}847$

left kernel	0.144s	13.57%
line sieve	0.351s	33.13%
matrix build	0.063s	5.93%
scan	0.016s	1.49%
smooth check	0.469s	44.28%
sqrt	0.017s	1.60%
total	1.058s	

Smooth check

Smoothness checking is expensive, and the cost is determined by the number of false positives. There is a tension between *cutoff* we calculated being too small (more false positives) and being too large (missing potentially smooth points). Accurately calculating the cutoff is important.

Initial calculation:

```
let cutoff = (num.to_f64().unwrap().log2() * 3.0 / 5.0).
    to_f32().unwrap();
```

Better cutoff

Better cutoff and scan:

```
let recalculate_every = 65536;
let fudge = 5.0;
for j in (0..area).step_by(recalculate_every) {
    let start = BigInt::from(j + sqrt_num.clone()) *
        BigInt::from(j + sqrt_num.clone()) - num.clone();
    let cutoff = start.to_f64().unwrap().log2() as f32 -
        fudge;
    for i in j..((j + recalculate_every).min(area)) {
        if sieve[i] > cutoff {
            smooth_points.push(i)
        }
    }
}
```

120 with better cutoff

Timing for 120-bit number, basic, own LA, better cutoff, M2

Parameters: sieve area: $50{,}000{,}000$, fb size: $28{,}000$

Matrix size: $1{,}502 \times 1{,}653$

left kernel	0.050s	6.55%
line sieve	0.335s	44.04%
matrix build	0.037s	4.80%
scan	0.022s	2.91%
smooth check	0.299s	39.37%
sqrt	0.018s	2.34%
total	0.761s	

Line sieve part deux

Our line sieve earlier was simple.

```
fn line_sieve(area: usize, sieve: &mut Vec<f32>, p: &u64,
    sqrtp: i64, logp: f32, roots: Vec<u64>) {
  for r in roots.iter() {
      let mut idx_i64 = (*r as i64 - sqrtp) % *p as i64;
      while idx_i64 < 0 {
          idx_i64 += *p as i64;
      }
      let mut idx = idx_i64 as usize;
      while idx < area {
          sieve[idx] += logp;
          idx += *p as usize;
      }
  }
}
```

Line sieve in pairs

For Math Reasons™, roots basically always come in pairs.

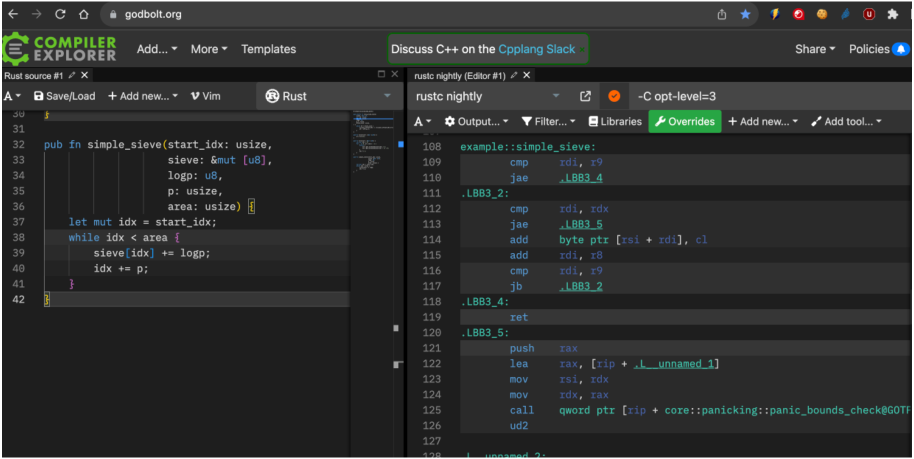
```
if roots.len() == 2 {
  let i0 = (roots[0] as isize - sqrtp) % p;
  let i1 = (roots[1] as isize - sqrtp) % p;
  let (i0, i1) = (i0.min(i1), i0.max(i1));
  unsafe {
      line_sieve_double(sieve.as_mut_ptr(), i0, i1, logp,
           p, area as isize);
  }
  return;
}
```

Line sieve in pairs (cont'd.)

```
pub unsafe fn line_sieve_double(sieve: *mut f32, mut i0:
    isize, mut i1: isize, logp: f32, p_i: isize,
    block_size: isize) {
    let sieve0 = sieve.offset(i0 - i1);
    let sieve1 = sieve;
    let init_i1 = i1;
    while i1 < block_size {
        *sieve0.offset(i1) += logp;
        *sieve1.offset(i1) += logp;
        i1 += p_i;
    }
    i0 += i1 - init_i1;
    while i0 < block_size {
        *sieve1.offset(i0) += logp;
        i0 += p_i;
    }
}
```

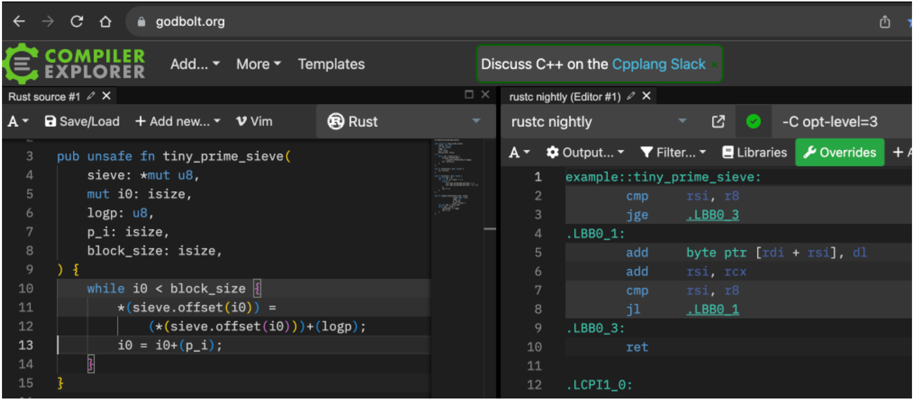
Why unsafe?

Safe



godbolt.org
COMPILER EXPLORER
Add...
More
Templates
Discuss C++ on the Cpplang Slack
Share
Policies
Rust source #1
Save/Load
+ Add new...
Vim
Rust
}
pub fn simple_sieve(start_idx: usize,
sieve: &mut [u8],
logp: u8,
p: usize,
area: usize) {
let mut idx = start_idx;
while idx < area {
sieve[idx] += logp;
idx += p;
}
}
rustc nightly (Editor #1)
rustc nightly
-C opt-level=3
Output...
Filter...
Libraries
Overrides
+ Add new...
Add tool...
example::simple_sieve:
cmp rdi, r9
jae .LBB3_4
.LBB3_2:
cmp rdi, rdx
jae .LBB3_5
add byte ptr [rsi + rdi], cl
add rdi, r8
cmp rdi, r9
jb .LBB3_2
.LBB3_4:
ret
.LBB3_5:
push rax
lea rax, [rip + .L__unnamed_1]
mov rsi, rdx
mov rdx, rax
call qword ptr [rip + core::panicking::panic_bounds_check@GOTP
ud2

Unsafe


godbolt.org
COMPILER EXPLORER
Add...
More
Templates
Discuss C++ on the Cpplang Slack
Rust source #1
Save/Load
Add new...
Vim
Rust
rustc nightly (Editor #1)
rustc nightly
-C opt-level=3
Output...
Filter...
Libraries
Overrides
pub unsafe fn tiny_prime_sieve(
sieve: *mut u8,
mut i0: isize,
logp: u8,
p_i: isize,
block_size: isize,
) {
while i0 < block_size {
*(sieve.offset(i0)) =
(*(sieve.offset(i0)))+(logp);
i0 = i0+(p_i);
}
}
example::tiny_prime_sieve:
cmp rsi, r8
jge .LBB0_3
.LBB0_1:
add byte ptr [rdi + rsi], dl
add rsi, rcx
cmp rsi, r8
jl .LBB0_1
.LBB0_3:
ret
.LCPI1_0:

120 with double sieve

Timing for 120-bit number, basic, own LA, double sieve, M2

Parameters: sieve area: $50{,}000{,}000$, fb size: $28{,}000$

Matrix size: $1{,}488 \times 1{,}768$

left kernel	0.056s	8.04%
line sieve	0.276s	39.32%
matrix build	0.037s	5.28%
scan	0.022s	3.08%
smooth check	0.301s	42.99%
sqrt	0.009s	1.28%
total	0.701s	

A little better.

Cache

Let's talk about memory.

High-end desktop processor

Kind	Size	Latency	Bandwidth
L1	32/32 KB	1 ns	400 GB/s
L2	512 KB	3 ns	150 GB/s
L3	128 MB	10 ns	100 GB/s
RAM	64 GB	75 ns	60 GB/s

Sieve size is $50{,}000{,}000 \cdot 4 = 200$ MB

M2 Cache

M2 Max

Kind	Size	Latency	Bandwidth
L1	128/192 KB	1 ns	1? TB/s
L2	32 MB	5 ns	1? TB/s
L3	48 MB	15 ns	400? GB/s
RAM	32 GB	100 ns	400 GB/s

120 with byte entries

Timing for 120-bit number with bytes, M2

Parameters: sieve area: $50{,}000{,}000$, fb size: $28{,}000$

Matrix size: $1{,}466 \times 1{,}504$

left kernel	0.038s	6.64%
line sieve	0.176s	30.82%
matrix build	0.031s	5.39%
scan	0.022s	3.87%
smooth check	0.295s	51.75%
sqrt	0.009s	1.53%
total	0.570s	

A little better.

140 with byte entries

Timing for 140-bit number with bytes, M2

Parameters: sieve area: $250{,}000{,}000$, fb size: $38{,}000$

Matrix size: $1{,}908 \times 1{,}997$

left kernel	0.091s	5.63%
line sieve	0.897s	55.40%
matrix build	0.069s	4.26%
scan	0.109s	6.74%
smooth check	0.434s	26.80%
sqrt	0.019s	1.17%
total	1.620s	

L1 Sieving

We're sieving over such a large area.

We would do better if we could take advantage of the caches to concentrate our sieve.

L1 Sieving

msieve splits the sieve into two parts:
Create area $\div$ L1 "hash tables".

Medium Primes

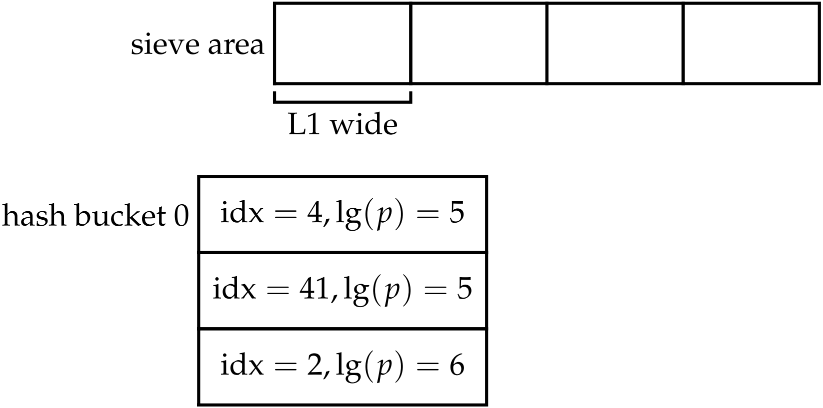
For each *p* is larger than the $3\times$ L1 cache size:

1. Calculate update as index *i*.
2. Add entry into "hash table" $(i \gg \text{L1 bits})$: $(i \,\&\, \text{L1 mask}, \log_2 p)$

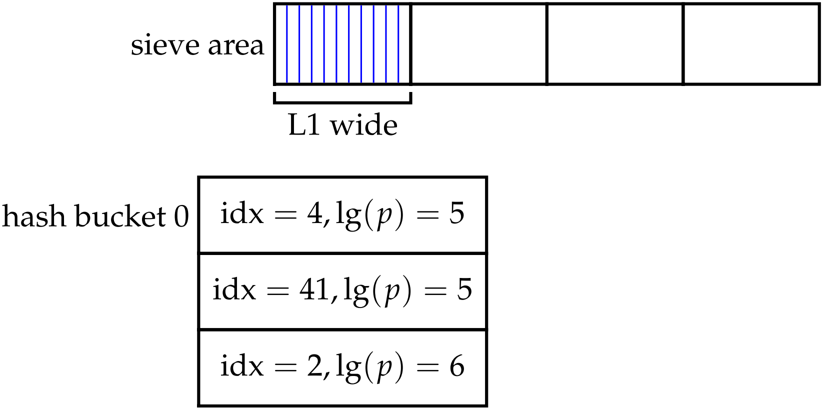
For each L1 block of the area

1. Sieve all primes *p* smaller than the $3\times$ L1 cache size, as normal.
2. Add in the updates for the hash table for this block.

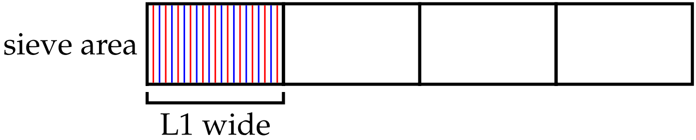
L1 Block Sieve



L1 Block Sieve



L1 Block Sieve



hash bucket 0

$\text{idx} = 4, \lg(p) = 5$
$\text{idx} = 41, \lg(p) = 5$
$\text{idx} = 2, \lg(p) = 6$

Timing with L1-aware

Timing for 140-bit number with L1-aware, M2

Parameters: sieve area: $250{,}000{,}000$, fb size: $38{,}000$
Matrix size: $1{,}943 \times 2{,}290$

left kernel	0.123s	11.19%
line sieve - alloc	0.027s	2.50%
line sieve - medium	0.000s	0.01%
line sieve - medium calculate	0.000s	0.00%
line sieve - small	0.149s	13.61%
matrix build	0.082s	7.51%
roots	0.048s	4.38%
scan	0.144s	13.18%
smooth check	0.506s	46.15%
sqrt	0.016s	1.47%
total	1.096s	

Timing with small allocation

Timing for 140-bit number with small alloc, M2

Parameters: sieve area: $250{,}000{,}000$, fb size: $38{,}000$
Matrix size: $1{,}946 \times 2{,}097$

left kernel	0.096s	9.23%
line sieve - alloc	0.000s	0.00%
line sieve - medium	0.000s	0.01%
line sieve - medium calculate	0.000s	0.00%
line sieve - small	0.178s	17.08%
matrix build	0.074s	7.14%
roots	0.054s	5.21%
scan	0.110s	10.60%
smooth check	0.467s	44.78%
sqrt	0.062s	5.95%
total	1.042s	

Smooth check improvements

When checking candidate number for smoothness:

1. Use `rug` (GMP) instead of `num_bigint`
2. Multiply the first several primes until they are $\approx 2^{64}$
3. Use GCD to pull those small primes off
4. At any point, can quickly check if the number is a known prime and stop.
5. Trial divide the rest
6. Quickly switch to 64-bit version once number falls below 2^{64}.

Small factor

Timing for 140-bit number with better small factor, M2

Parameters: sieve area: $250{,}000{,}000$, fb size: $38{,}000$

Matrix size: $1{,}935 \times 2{,}134$

left kernel	0.104s	19.20%
line sieve - alloc	0.000s	0.00%
line sieve - medium	0.000s	0.01%
line sieve - medium calculate	0.000s	0.00%
line sieve - small	0.180s	33.26%
matrix build	0.075s	13.80%
roots	0.054s	10.01%
scan	0.115s	21.21%
smooth check	0.011s	1.97%
sqrt	0.003s	0.53%
total	0.542s	

Linear Algebra

We've been mostly ignoring the linear algebra stuff, as it's good enough.

It will mostly be good enough even as we scale up.

We could maybe do better than the `n_algebra::DMatrix<u8>` though.

Rust has `portable_simd`, which allows us to have SIMD types, such as a 256-bit type `u64x4` that acts like `[u64; 4]`.

Linear Algebra (M2 Max Timings)

Timings for 11000×11000 matrix solve (from 220-bit number).

nalgebra::DMatrix<u8>

left kernel: 19.738142 seconds
matrix build: 0.216748 seconds

Linear Algebra (M2 Max Timings)

Timings for 11000×11000 matrix solve (from 220-bit number).

`nalgebra::DMatrix<u8>`

left kernel: 19.738142 seconds

matrix build: 0.216748 seconds

`BinaryMatrix64`

left kernel: 2.370887 seconds

matrix build: 0.049763 seconds

Linear Algebra (M2 Max Timings)

Timings for 11000×11000 matrix solve (from 220-bit number).

`nalgebra::DMatrix<u8>`

left kernel: 19.738142 seconds

matrix build: 0.216748 seconds

`BinaryMatrix64`

left kernel: 2.370887 seconds

matrix build: 0.049763 seconds

`BinaryMatrixSimd<u64x64>`

left kernel: 2.519738 seconds

matrix build: 0.051588 seconds

SIMD Scanning

SIMD didn't help us much with linear algebra, but another place we do a lot of repetitive, simple memory stuff is scanning.
Before:

```
let mut smooth_points = vec![];
let cutoff = calculate_cutoff(poly, center, add_to_point,
    large_prime_bound, log_scale);
for i in 0..sieve.len() {
    if sieve[i] >= cutoff {
        smooth_points.push(i + add_to_point);
    }
}
timers.record("scan", scan_start);
```

SIMD Scanning (cont'd.)

After:

```
let cutoff = calculate_cutoff(poly, center, add_to_point,
    large_prime_bound, log_scale);
let sieved = sieve.as_ptr() as *const Simd<u8, LANES>;
let mask = Simd::<u8, LANES>::splat(cutoff);
for i in 0..sieve.len() >> LANES.trailing_zeros() {
    let x = unsafe { (*(sieved.offset(i as isize))).
        simd_ge(mask) };
    if x.any() {
        for k in 0..LANES {
            unsafe {
                if x.test_unchecked(k) {
                    smooth_points.push((i << LANES.
                        trailing_zeros()) + k +
                        add_to_point);
                }
            }
        }
    }
```

Scan comparison

Again, for a 220-bit number:

Non-SIMD

```
scan       : 5.216741 seconds (18.83%) count 122854
```

SIMD

```
scan u8x16: 0.479979 seconds ( 1.97%) count 126502
```

Where did we get to?

Running on the latest M2 Max processor.

1. 100 bits: 0.009 seconds $\pm$ 14%
2. 120 bits: 0.031 seconds $\pm$ 14%
3. 140 bits: 0.094 seconds $\pm$ 12%
4. 160 bits: 0.257 seconds $\pm$ 1%
5. 180 bits: 1.576 seconds $\pm$ 52%
6. 200 bits: 8.877 seconds $\pm$ 40%
7. 220 bits: 33.11 seconds $\pm$ 31%
8. 240 bits: 139.4 seconds $\pm$ 23%

Side note: msieve optimization

It seemed only fair to do a basic optimization for `msieve` so it can use a 128-kB L1 cache, nearly doubling its speed from the 32-kB default.

	msieve-arm64-32kb.txt	msieve-arm64-64kb.txt		msieve-arm64-128kb.txt	
	sec/op	sec/op	vs base	sec/op	vs base
Msieve80-12	5.062m ± 1%	5.134m ± 1%	+1.41% (p=0.000 n=10)	5.114m ± 2%	~ (p=0.105 n=10)
Msieve100-12	21.40m ± 2%	21.56m ± 2%	~ (p=0.218 n=10)	21.38m ± 1%	~ (p=0.842 n=10+9)
Msieve120-12	29.66m ± 1%	29.90m ± 1%	~ (p=0.579 n=10)	29.66m ± 2%	~ (p=0.393 n=10)
Msieve140-12	63.09m ± 3%	64.22m ± 3%	~ (p=0.190 n=10)	62.58m ± 3%	~ (p=0.481 n=10)
Msieve160-12	128.5m ± 6%	120.1m ± 4%	-6.59% (p=0.002 n=10)	114.4m ± 6%	-10.97% (p=0.000 n=10)
Msieve180-12	482.2m ± 10%	385.1m ± 6%	-20.14% (p=0.000 n=10)	358.9m ± 6%	-25.56% (p=0.000 n=10)
Msieve200-12	2.152 ± 19%	1.669 ± 21%	-22.47% (p=0.009 n=10)	1.421 ± 22%	-33.98% (p=0.000 n=10)
Msieve220-12	9.782 ± 21%	7.171 ± 13%	-26.69% (p=0.000 n=10)	5.452 ± 15%	-44.26% (p=0.000 n=10)
Msieve240-12	31.36 ± 20%	27.87 ± 14%	~ (p=0.063 n=10)	22.77 ± 25%	-27.39% (p=0.000 n=10)
geomean	272.5m	245.8m	-9.80%	224.9m	-17.49%

How well did we do?

What's good?

- Faster than `msieve` for small sizes, probably due to better parameter tuning.
- Within a factor of 4 or so of (slightly optimized) `msieve` for larger sizes.

What's missing?

- `msieve` does way better linear algebra.
- `msieve` has better line sieve code tuned for x86.
- Does a better job at combining relations.

Lessons

Rust wins:

- Rust (and LLVM) are solid, performance-wise.
- Auto-vectorization has come a long way, though it does a terrible job sometimes. (See `&[u8].fill()`.)
- Rust makes modularity easier.
- Rust is fun.

Rust losses:

- Rust has a brutal learning curve.
- I still feel like I have to do half the compiler's job for it.
- Rust memory management is confusing, and encourages bad behavior. For example, using `fn xyz() -> SomeType` instead of `fn xyz() -> &SomeType` when `SomeType` is really large, just to avoid the borrow checker.

Further reading

- Book: *Prime Numbers: A Computational Perspective* (2005) by Richard Crandall and Carl Pomerance.
- Code: `msieve`.
- `https://godbolt.org`.
- `https://en.wikipedia.org/wiki/Quadratic_sieve` and its references.
- These slides and code: `https://swenson.io/speaking.html`