The Power of Hashing with Mersenne Primes

Thomas Dybdahl Ahle[1], Jakob Tejs Bæk Knudsen[2], and Mikkel Thorup[2]

[1]Facebook, BARC, *thomas@ahle.dk*
[2]University of Copenhagen, BARC, {*jakn, mthorup*}*@di.ku*

May 6, 2021

Abstract

The classic way of computing a k-universal hash function is to use a random degree-$(k-1)$ polynomial over a prime field $\mathbb{Z}_p$. For a fast computation of the polynomial, the prime p is often chosen as a Mersenne prime $p = 2^b - 1$.

In this paper, we show that there are other nice advantages to using Mersenne primes. Our view is that the hash function's output is a b-bit integer that is uniformly distributed in $\{0, \dots, 2^b - 1\}$, except that p (the all 1s value in binary) is missing. Uniform bit strings have many nice properties, such as splitting into substrings which gives us two or more hash functions for the cost of one, while preserving strong theoretical qualities. We call this trick "Two for one" hashing, and we demonstrate it on 4-universal hashing in the classic Count Sketch algorithm for second-moment estimation.

We also provide a new fast branch-free code for division and modulus with Mersenne primes. Contrasting our analytic work, this code generalizes to any Pseudo-Mersenne primes $p = 2^b - c$ for small c.

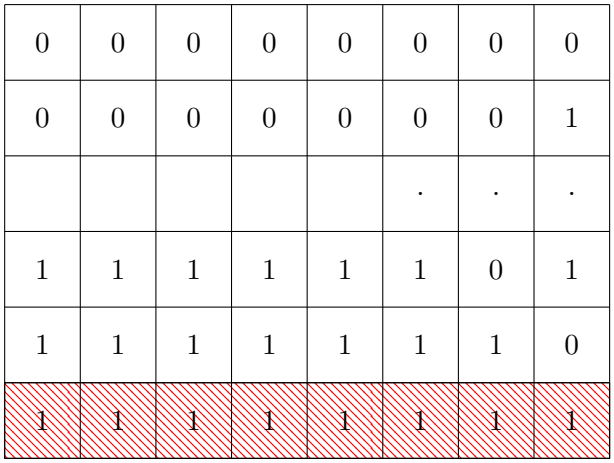

Figure 1: The output of a random polynomial modulo $p = 2^b - 1$ is uniformly distributed in $[p]$, so each bit has the same distribution, which is only $1/p$ biased towards 0.

1 Introduction

The classic way to implement k-universal hashing is to use a random degree $(k-1)$-polynomial over a finite field [WC81]. Mersenne primes, which are prime numbers on the form $2^b - 1$, have been used to implement finite fields efficiently for more than 40 years using standard portable code [CW79].

The speed of hashing is important because it is often an inner-loop bottle-neck in data analysis. A good example is when hashing is used in the sketching of high volume data streams, such as traffic through an Internet router, and then this speed is critical to keep up with the stream. A running example in this paper is the classic second moment estimation using 4-universal hashing in count sketches [CCF04]. Count Sketches are linear maps that statistically preserve the Euclidean norm. They are also popular in machine learning under the name "Feature Hashing" [Moo89, WDL$^+$09].

In this paper, we argue that uniform random values from Mersenne prime fields are not only fast to compute but *have special advantages different from any other field.* While it is natural to consider values mod $p = 2^b - 1$ as "nearly" uniform b-bit strings (see Figure 1), we show that the small bias in our hash values can usually be turned into an advantage. In particular our analysis justify splitting single hash values into two or more for a significant computational speed-up, what we call the "Two for one" trick.

We also show that while the $1/p$ bias of such strings would usually result in relative errors of order n/p for Count Sketch, a specialized analysis yields relative errors of just n/p^2. The analysis is based on simple moments, and give similar improvements for any algorithm analyzed this way. Loosely speaking, this means that we for a desired small error can reduce the bit-length of the primes to less than half. This saves not only space, it means that we can speed up the multiplications with a factor of 2.

Finally we provide a fast, simple and branch-free algorithm for division and modulus with Mersenne primes. Contrasting our analytic work, this code generalizes to so-called Pseudo-Mersenne primes [VTJ14] of the form $p = 2^b - c$ for small c. Our new code is simpler and faster than the classical algorithm of Crandall [Cra92].

We provide experiments of both algorithms in Section 5. For the rest of the introduction we will give a more detailed review of the new results.

1.1 Hashing uniformly into b bits

A main point in this paper is that having hash values uniform in $[2^b - 1] = \{0, \ldots, 2^b - 2\}$ is almost as good as having uniform b-bit strings, but of course, it would be even better if we just had uniform b-bit strings.

We do have the fast multiply-shift scheme of Dietzfelbinger [Die96], which directly gives 2-universal hashing from b-bit strings to ℓ-bit strings, but for $k > 2$, there is no such fast k-universal hashing scheme that can be implemented with standard portable code.

More recently it has been suggested to use carry-less multiplication for k-universal hashing into bit strings (see, e.g., Lemire [LK14]) but contrasting the hashing with Mersenne primes, this is less standard (takes some work to get it to run on different computers) and slower (by about 30-50% for larger k on the computers we tested in Section 5). Moreover, the code for different bit-lengths b is quite different because we need quite different irreducible polynomials.

Another alternative is to use tabulation based methods which are fast but use a lot of space [Sie04, Tho13], that is, space $s = 2^{\Omega(b)}$ to calculate k-universal hash function in constant time from b-bit keys to ℓ-bit hash values. The large space can be problematic.

A classic example where constant space hash functions are needed is in static two-level hash functions [FKS84]. To store n keys with constant access time, you use n second-level hash tables, each with its own hash function. Another example is small sketches such as the Count Sketch [CCF04] discussed in this paper. Here we may want to store the hash function as part of the sketch, e.g., to query the value of a given key. Then the hash value has to be directly computable from the small representation, ruling out tabulation based methods (see further explanation at the end of Section 1.3.1).

It can thus be problematic to get efficient k-universal hashing directly into b-bit strings, and this is why we in this paper analyse the hash values from Mersenne prime fields that are much easier to generate.

1.2 Polynomial hashing using Mersenne primes

Before discussing the special properties of Mersenne primes in algorithm analysis, we show how they are classically used to do fast field computations, and propose a new simple algorithm for further speed-ups in the hashing case.

The definition of k-universal hashing goes back to Carter and Wegman [WC81].

Definition 1. *A random hash function $h : U \to R$ is k-universal if for k distinct keys $x_0, \ldots, x_{k-1} \in U$, the k-tuple $(h(x_0), \ldots, h(x_{k-1}))$ is uniform in R^k.*

Note that the definition also implies the values $h(x_0), \ldots, h(x_{k-1})$ are independent. A very similar concept is that of k-independence, which has only this requirement but doesn't include that values must be uniform.

For $k > 2$ the standard k-universal hash function is uniformly random degree-$(k-1)$ polynomial over a prime field $\mathbb{Z}_p$, that is, we pick a uniformly random vector $\vec{a} = (a_0, \ldots, a_{k-1}) \in \mathbb{Z}_p^k$ of k coefficients, and define $h_{\vec{a}} : [p] \to [p]$, [1] by

$$h_{\vec{a}}(x) = \sum_{i \in [k]} a_i x^i \mod p.$$

Given a desired key domain $[u]$ and range $[r]$ for the hash values, we pick $p \geq \max\{u, r\}$ and define $h^r_{\vec{a}} : [u] \to [r]$ by

$$h^r_{\vec{a}}(x) = h_{\vec{a}}(x) \bmod r.$$

[1] We use the notation $[s] = \{0, \ldots, s-1\}$.

The hash values of k distinct keys remain independent while staying as close as possible to the uniform distribution on $[r]$. (This will turn out to be very important.)

In terms of speed, the main bottleneck in the above approach is the mod operations. If we assume $r = 2^\ell$, the mod r operation above can be replaced by a binary AND (&): $x \bmod r = x \,\&\, r - 1$. Similarly, Carter and Wegman [CW79] used a Mersenne prime $p = 2^b - 1$,[2] to speed up the computation of the (mod p) operations:

$$y \equiv y - \lfloor y/2^b \rfloor (2^b - 1) = (y \bmod 2^b) + \lfloor y/2^b \rfloor \pmod{p}. \tag{1}$$

Again allowing us to use the very fast bit-wise AND (&) and the right-shift (≫), instead of the expensive modulo operation.

Of course, (1) only reduces y to an equivalent value mod p, not to the smallest one, which is what we usually want. For this reason one typically adds a test "if $y \geq p$ then $y \leftarrow y - p$". We show an implementation in Algorithm 1 below with one further improvement: By assuming that $p = 2^b - 1 \geq 2u - 1$ (which is automatically satisfied in the typical case where u is a power of two, e.g., 2^{32} or 2^{64}) we can get away with only doing this test once, rather than at every loop. Note the proof by loop invariant in the comments.

Algorithm 1 For $x \in [u]$, prime $p = 2^b - 1 \geq 2u - 1$, and $\vec{a} = (a_0, \ldots, a_{k-1}) \in [p]^k$, computes $y = h_{\vec{a}}(x) = \sum_{i \in [k]} a_i x^i \mod p$.

```
y ← a_{k-1}
for i = q − 2, ..., 0 do                          ▷ Invariant:  y < 2p
    y ← y ∗ x + a_i                    ▷  y < 2p(u − 1) + (p − 1) < (2u − 1)p ≤ p^2
    y ← (y & p) + (y ≫ b)                          ▷  y < p + p^2/2^b < 2p
if y ≥ p then
    y ← y − p                                                    ▷ y < p
```

In Section 1.5 we will give one further improvement to Algorithm 1. In the next sections we will argue that Mersenne primes are not only fast, but have special properties not found in other finite fields.

1.2.1 Selecting arbitrary bits

If we had b uniform bits, we could partition them any way we'd like and get smaller independent strings of uniform random bits. The first property of random values modulo Mersenne primes we discuss is what happens when the same thing is done on a random value in $[2^b - 1]$ instead.

More formally, let $\mu : [2^b] \to [2^\ell]$ be any map that selects ℓ distinct bits, that is, for some $0 \leq j_1 < \cdots < j_\ell < b$, $\mu(y) = y_{j_1} \cdots y_{j_\ell}$. For example, if $j_i = i - 1$, then we are selecting the most significant bits, and then μ can be implemented as $y \mapsto y \gg (b - \ell)$. Alternatively, if $j_i = b - i$, then we are selecting the least significant bits, and then μ can be implemented as $y \mapsto y \,\&\, (2^\ell - 1) = y \,\&\, (r - 1)$.

We assume a k-universal hash function $h : [u] \to [p]$, e.g., the one from Algorithm 1. To get hash values in $[r]$, we use $\mu \circ h$. Since μ is deterministic, the hash values of up to k distinct keys remain independent with $\mu \circ h$. The issue is that hash values from $\mu \circ h$ are not quite uniform in $[r]$.

Recall that for any key x, we have $h(x)$ uniformly distributed in $[2^b - 1]$. This is the uniform distribution on b-bit strings except that we are missing $p = 2^b - 1$. Now p is the all 1s, and

[2] e.g., $p = 2^{61} - 1$ for hashing 32-bit keys or $p = 2^{89} - 1$ for hashing 64-bit keys.

$\mu(p) = r - 1$. Therefore

$$\text{for } i < r-1, \quad \Pr[\mu(h(x)) = i] = \lceil p/r \rceil / p = ((p+1)/r)/p = (1 + 1/p)/r \tag{2}$$

$$\text{and} \quad \Pr[\mu(h(x)) = r-1] = \lfloor p/r \rfloor / p = ((p+1-r)/r)/p = (1 - (r-1)/p)/r. \tag{3}$$

Thus $\Pr[\mu(h(x)) = i] \leq (1 + 1/p)/r$ for all $i \in [r]$. This upper-bound only has a relative error of $1/p$ from the uniform $1/r$.

Combining (2) and (3) with pairwise independence, for any distinct keys $x, y \in [u]$, we show that the collision probability is bounded

$$\begin{aligned} \Pr[\mu(h(x)) = \mu(h(y))] &= (r-1)((1+1/p)/r)^2 + ((1-(r-1)r/p)/r)^2 \\ &= (1 + (r-1)/p^2)/r. \end{aligned} \tag{4}$$

Thus the relative error r/p^2 is small as long as p is large.

The problem with non-Mersenne Primes Suppose $c \neq 1$ and we want to select arbitrary bits like in the arguments above. If we pick the least significant bits we get a generic upper bound of $(1 + c/p)/r$, which is not too bad for small c. Here there is no conceptual difference to our Mersenne results.

However take the opposite extreme where we pick just the one most significant bit and $c = 2^{b-1} - 1$ (so $p = 2^{b-1} + 1$, a Fermat prime). That bit is 0 with probability $1 - 1/p$ and 1 only with probability p – virtually a constant. We might try to fix this by xoring the output with a random number from $[2^b]$ (or add $C \in [2^b]$ and take mod 2^b), but that will only make the bits uniform, not actually dependent on the key. Thus if we hash two keys, x_1 and x_2, mod $2^{b-1} + 1$ and take the top bit from each one, *they will nearly always be the same, independent of whether* $x_1 = x_2$.

More generally, say we pick the ℓ most significant bits and $c \leq 2^{b-1} - 2^{b-\ell}$, then $2^{b-\ell}$ elements from $[p]$ map to 0 while only $\max\{0, 2^{b-\ell} - c\}$ map to the all 1s. More concretely, take $\ell = b/2$ and $c = 2^{b/2} \approx \sqrt{p}$ (typical for generalized Mersenne primes) *then the top ℓ bits hit the all 1s with 0 probability*, while the all 0s is twice as common as the remaining values.

1.3 Two-for-one hash functions in second moment estimation

In this section, we discuss how we can get several hash functions for the price of one, and apply the idea to second moment estimation using Count Sketches [CCF04].

Suppose we had a k-universal hash function into b-bit strings. We note that using standard programming languages such as C, we have no simple and efficient method of computing such hash functions when $k > 2$. However, later we will argue that polynomial hashing using a Mersenne prime $2^b - 1$ delivers a better-than-expected approximation.

Let $h : U \to [2^b]$ be k-universal. By definition this means that if we have $j \leq k$ distinct keys $x_0, \ldots, x_{j-1}$, then $(h(x_0), \ldots, h(x_{j-1}))$ is uniform in $[2^b]^j \equiv [2]^{bj}$, so this means that *all* the bits in $h(x_0), \ldots, h(x_{j-1})$ are independent and uniform. We can use this to split our b-bit hash values into smaller segments, and sometimes use them as if they were the output of universally computed hash functions.

We illustrate this idea below in the context of the second moment estimation. For this purpose the "split" we will be considering is into the first bit and the remaining bits. [3]

[3] Note there are other ways to construct this sketch, which only use one hash function, such as [TZ12]. The following should thus not be taken as "the only way" to achieve this result, but as an example of how the "intuitive approach" turns out to work when hashing with Mersenne primes.

1.3.1 Second moment estimation

We now review the second moment estimation of streams based on Count Sketches [CCF04] (which are based on the celebrated second moment AMS-estimator from [AMS99].)

The basic setup is as follows: For keys in $[u]$ and integer values in $\mathbb{Z}$, we are given a stream of key/value $(x_0, \Delta_0), \ldots, (x_{n-1}, \Delta_{n-1}) \in [u] \times \mathbb{Z}$. The total value of key $x \in [u]$ is

$$f_x = \sum_{i \in [n], x_i = x} \Delta_i.$$

We let $n \leq u$ be the number of non-zero values $f_x \neq 0$, $x \in [u]$. Often n is much smaller than u. We define the mth moment $F_m = \sum_{x \in [u]} f_y^m$. The goal here is to estimate the second moment $F_2 = \sum_{x \in [u]} f_x^2 = \|f\|_2^2$.

Algorithm 2 Count Sketch. Uses a vector/array C of r integers and two independent 4-universal hash functions $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$.

procedure INITIALIZE
  For $i \in [t]$, set $C[i] \leftarrow 0$.
procedure PROCESS(x, Δ)
  $C[i(x)] \leftarrow C[i(x)] + s(x)\Delta$.
procedure OUTPUT
  return $\sum_{i \in [t]} C[i]^2$.

The standard analysis [CCF04] shows that

$$\mathrm{E}[X] = F_2 \tag{5}$$

$$\mathrm{Var}[X] = 2(F_2^2 - F_4)/r < 2F_2^2/r \tag{6}$$

We see that by choosing larger and larger r we can make X concentrate around $F_2 = \|f\|_2^2$. Here $X = \sum_{i \in [r]} C[i]^2 = \|C\|_2^2$. Now C is a randomized function of f, and as r grows, we get $\|C(f)\|_2^2 \approx \|f\|_2^2$, implying $\|C(f)\|_2 \approx \|f\|_2$, that is, the Euclidean norm is statistically preserved by the Count Sketch. However, the Count Sketch is also a linear function, so Euclidean distances are statistically preserved, that is, for any $f, g \in \mathbb{Z}^u$,

$$\|f - g\|_2 \approx \|C(f - g)\|_2 = \|C(f) - C(g)\|_2.$$

Thus, when we want to find close vectors, we can just work with the much smaller Count Sketches. The count sketch C can also be used to estimate any single value f_x. To do this, we use the unbiased estimator $X_x = s(x)C[i(x)]$. This is yet another standard use of count sketch [CCF04]. It requires direct access to both the sketch C and the two hash functions s and i. To get concentration one takes the median of multiple such estimators.

1.3.2 Two-for-one hash functions with b-bit hash values

As the count sketch is described above, it uses two independent 4-universal hash functions $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$, but 4-universal hash functions are generally slow to compute, so, aiming to save roughly a factor 2 in speed, a tempting idea is to compute them both using a single hash function.

The analysis behind (5) and (6) does not quite require $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$ to be independent. It suffices that the hash values are uniform and that for any given set of

$j \leq 4$ distinct keys $x_0, \ldots, x_{j-1}$, the $2j$ hash values $i(x_0), \ldots, i(x_{j-1}), s(x_0), \ldots, s(x_{j-1})$ are independent. A critical step in the analysis is that if a value A depends on the first $j-1$ values ($A = A(i(x_0), \ldots, i(x_{j-1}), s(x_1), \ldots, s(x_{j-1}))$), but doesn't depend on $s(x_0)$, then

$$\mathrm{E}[s(x_0)A] = 0. \tag{7}$$

This follows because $\mathrm{E}[s(x_0)] = 0$ by uniformity of $s(x_0)$ and because $s(x_0)$ is independent of A.

Assuming that $r = 2^\ell$ is a power of two, we can easily construct $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$ using a single 4-universal hash function $h : [u] \to [2^b]$ where $b > \ell$. Recall that all the bits in $h(x_0), \ldots, h(x_3)$ are independent. We can therefore use the ℓ least significant bits of $h(x)$ for $i(x)$ and the most significant bit of $h(x)$ for a bit $a(x) \in [2]$, and finally set $s(x) = 1 - 2a(x)$. It is then easy to show that if h is 4-universal then h satisfies eq. (7).

Algorithm 3 For key $x \in [u]$, compute $i(x) = i_x \in [2^\ell]$ and $s(x) = s_x \in \{-1, 1\}$, using $h : [u] \to [2^b]$ where $b > \ell$.

$h_x \leftarrow h(x)$	$\triangleright$ h_x uses b bits
$i_x \leftarrow h_x \mathbin{\&} (2^\ell - 1)$	$\triangleright$ i_x gets ℓ least significant bits of h_x
$a_x \leftarrow h_x \gg (b-1)$	$\triangleright$ a_x gets the most significant bit of h_x
$s_x \leftarrow 1 - (a_x \ll 1)$	$\triangleright$ $a_x \in [2]$ is converted to a sign $s_x \in \{-1, 1\}$

Note that Algorithm 3 is well defined as long as h returns a b-bit integer. However, eq. (7) requires that h is k-universal into $[2^b]$, which in particular implies that the hash values are uniform in $[2^b]$.

1.3.3 Two-for-one hashing with Mersenne primes

Above we discussed how useful it would be with k-universal hashing mapping uniformly into b-bit strings. The issue was that the lack of efficient implementations with standard portable code if $k > 2$. However, when $2^b - 1$ is a Mersenne prime $p \geq u$, then we do have the efficient computation from Algorithm 1 of a k-universal hash function $h : [u] \to [2^b - 1]$. The hash values are b-bit integers, and they are uniformly distributed, except that we are missing the all 1s value $p = 2^b - 1$. We want to understand how this missing value affects us if we try to split the hash values as in Algorithm 3. Thus, we assume a k-universal hash function $h : [u] \to [2^b - 1]$ from which we construct $i : [u] \to [2^\ell]$ and $s : [u] \to \{-1, 1\}$ as described in Algorithm 3. As usual, we assume $2^\ell > 1$. Since i_x and s_x are both obtained by selection of bits from h_x, we know from Section 1.2.1 that each of them have close to uniform distributions. However, we need a good replacement for (7) which besides uniformity, requires i_x and s_x to be independent, and this is certainly not the case.

Before getting into the analysis, we argue that we really do get two hash functions for the price of one. The point is that our efficient computation in Algorithm 1 requires that we use a Mersenne prime $2^b - 1$ such that $u \leq 2^{b-1}$, and this is even if our final target is to produce just a single bit for the sign function $s : [u] \to \{-1, 1\}$. We also know that $2^\ell < u$, for otherwise we get perfect results implementing $i : [u] \to [2^\ell]$ as the identity function (perfect because it is collision-free). Thus we can assume $\ell < b$, hence that h provides enough bits for both s and i.

We now consider the effect of the hash values from h being uniform in $[2^b - 1]$ instead of in $[2^b]$. Suppose we want to compute the expected value of an expression B depending only on the independent hash values $h(x_0), \ldots, h(x_{j-1})$ of $j \leq k$ distinct keys $x_0, \ldots, x_{j-1}$.

Our generic idea is to play with the distribution of $h(x_0)$ while leaving the distributions of the other independent hash values $h(x_0) \ldots, h(x_{j-1})$ unchanged, that is, they remain uniform

in $[2^b - 1]$. We will consider having $h(x_0)$ uniformly distributed in $[2^b]$, denoted $h(x_0) \sim \mathcal{U}[2^b]$, but then we later have to subtract the "fake" case where $h(x_0) = p = 2^b - 1$. Making the distribution of $h(x_0)$ explicit, we get

$$\begin{aligned}\mathop{\mathrm{E}}_{h(x_0)\sim\mathcal{U}[p]}[B] &= \sum_{y\in[p]} \mathrm{E}[B \mid h(x_0) = y]/p \\ &= \sum_{y\in[2^b]} \mathrm{E}[B \mid h(x_0) = y]/p - \mathrm{E}[B \mid h(x_0) = p]/p \\ &= \mathop{\mathrm{E}}_{h(x_0)\sim\mathcal{U}[2^b]}[B](p+1)/p - \mathrm{E}[B \mid h(x_0) = p]/p.\end{aligned} \tag{8}$$

Let us now apply this idea our situation where $i : [u] \to [2^\ell]$ and $s : [u] \to \{-1, 1\}$ are constructed from h as described in Algorithm 3. We will prove

Lemma 1.1. *Consider distinct keys $x_0, \ldots, x_{j-1}$, $j \le k$ and an expression $B = s(x_0)A$ where A depends on $i(x_0), \ldots, i(x_{j-1})$ and $s(x_1), \ldots, s(x_{j-1})$ but not $s(x_0)$. Then*

$$\mathrm{E}[s(x_0)A] = \mathrm{E}[A \mid i(x_0) = 2^\ell - 1]/p. \tag{9}$$

Proof. When $h(x_0) \sim \mathcal{U}[2^b]$, then $s(x_0)$ is uniform in $\{-1, 1\}$ and independent of $i(x_0)$. The remaining $(i(x_i), s(x_i))$, $i \ge 1$, are independent of $s(x_0)$ because they are functions of $h(x_i)$ which is independent of $h(x_0)$, so we conclude that

$$\mathop{\mathrm{E}}_{h(x)\sim\mathcal{U}[2^b]}[s(x_0)A] = 0$$

Finally, when $h(x_0) = p$, we get $s(x_0) = -1$ and $i(x_0) = 2^\ell - 1$, so applying (8), we conclude that

$$\mathrm{E}[s(x_0)A] = -\mathrm{E}[s(x_0)A \mid h(x_0) = p]/p = \mathrm{E}[A \mid i(x_0) = 2^\ell - 1]/p.$$

□

Above (9) is our replacement for (7), that is, when the hash values from h are uniform in $[2^b - 1]$ instead of in $[2^b]$, then $\mathrm{E}[s(x_0)B]$ is reduced by $\mathrm{E}[B \mid i(x_0) = 2^\ell - 1]/p$. For large p, this is a small additive error. Using this in a careful analysis, we will show that our fast second moment estimation based on Mersenne primes performs almost perfectly:

Theorem 1.2. *Let $r > 1$ and $u > r$ be powers of two and let $p = 2^b - 1 > u$ be a Mersenne prime. Suppose we have a 4-universal hash function $h : [u] \to [2^b - 1]$, e.g., generated using Algorithm 1. Suppose $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$ are constructed from h as described in Algorithm 3. Using this i and s in the Count Sketch Algorithm 2, the second moment estimate $X = \sum_{i\in[k]} C_i^2$ satisfies:*

$$\mathrm{E}[X] < (1 + n/p^2)\, F_2, \quad |\mathrm{E}[X] - F_2| \le F_2(n-1)/p^2, \quad \mathrm{Var}[X] < 2F_2^2/r.$$

The difference from (5) and (6) is negligible when p is large. Theorem 1.2 will be proved in Section 2.

Recall our discussion from the end of Section 1.2.1. If we instead had used the b-bit prime $p = 2^{b-1} + 1$, then the sign-bit a_x would be extremely biased with $\Pr[a_x = 0] = 1 - 1/p$ while $\Pr[a_x = 1] = 1/p$, leading to extremely poor performance.

1.4 An arbitrary number of buckets

We now consider the general case where we want to hash into a set of buckets R whose size is not a power of two. Suppose we have a 2-universal hash function $h : U \to Q$. We will compose h with a map $\mu : Q \to R$, and use $\mu \circ h$ as a hash function from U to R. Let $q = |Q|$ and $r = |R|$. We want the map μ to be *most uniform* in the sense that for bucket $i \in R$, the number of elements from Q mapping to i is either $\lfloor q/r \rfloor$ or $\lceil q/r \rceil$. Then the uniformity of hash values with h implies for any key x and bucket $i \in R$

$$\lfloor q/r \rfloor / q \leq \Pr[\mu(h(x)) = i] \leq \lceil q/r \rceil / q.$$

Below we typically have $Q = [q]$ and $R = [r]$. A standard example of a most uniform map $\mu : [q] \to [r]$ is $\mu(x) = x \bmod r$ which the one used above when we defined $h^r : [u] \to [r]$, but as we mentioned before, the modulo operation is quite slow unless r is a power of two.

Another example of a most uniform map $\mu : [q] \to [r]$ is $\mu(x) = \lfloor xr/q \rfloor$, which is also quite slow in general, but if $q = 2^b$ is a power of two, it can be implemented as $\mu(x) = (xr) \gg b$ where $\gg$ denotes right-shift. This would be yet another advantage of having k-universal hashing into $[2^b]$.

Now, our interest is the case where q is a Mersenne prime $p = 2^b - 1$. We want an efficient and most uniform map $\mu : [2^b - 1]$ into any given $[r]$. Our simple solution is to define

$$\mu(v) = \lfloor (v+1)r/2^b \rfloor = ((v+1)r) \gg b. \tag{10}$$

Lemma 1.3 (iii) below states that (10) indeed gives a most uniform map.

Lemma 1.3. *Let r and b be positive integers. Then*

(i) $v \mapsto (vr) \gg b$ *is a most uniform map from* $[2^b]$ *to* $[r]$.

(ii) $v \mapsto (vr) \gg b$ *is a most uniform map from* $[2^b] \setminus \{0\} = \{1, \ldots, 2^b - 1\}$ *to* $[r]$.

(iii) $v \mapsto ((v+1)r) \gg b$ *is a most uniform map from* $[2^b - 1]$ *to* $[r]$.

Proof. Trivially (ii) implies (iii). The statement (i) is folklore and easy to prove, so we know that every $i \in [r]$ gets hit by $\lfloor 2^b/r \rfloor$ or $\lceil 2^b/r \rceil$ elements from $[2^b]$. It is also clear that $\lceil 2^b/r \rceil$ elements, including 0, map to 0. To prove (ii), we remove 0 from $[2^b]$, implying that only $\lceil 2^b/r \rceil - 1$ elements map to 0. For all positive integers q and r, $\lceil (q+1)/r \rceil - 1 = \lfloor q/r \rfloor$, and we use this here with $q = 2^b - 1$. It follows that all buckets from $[r]$ get $\lfloor q/r \rfloor$ or $\lfloor q/r \rfloor + 1$ elements from $Q = \{1, \ldots, q\}$. If r does not divide q then $\lfloor q/r \rfloor + 1 = \lceil q/r \rceil$, as desired. However, if r divides q, then $\lfloor q/r \rfloor = q/r$, and this is the least number of elements from Q hitting any bucket in $[r]$. Then no bucket from $[r]$ can get hit by more than $q/r = \lceil q/r \rceil$ elements from Q. This completes the proof of (ii), and hence of (iii). □

We note that our trick does not work when $q = 2^b - c$ for $c \geq 2$, that is, using $v \mapsto ((v+c)r) \gg b$, for in this general case, the number of elements hashing to 0 is $\lceil 2^b/r \rceil - c$, or 0 if $c \geq \lfloor 2^b/r \rfloor$. One may try many other hash functions $(c_1 vr + c_2 v + c_3 r + c_4) \gg b$ similarly without any luck. Our new uniform map from (10) is thus very specific to Mersenne prime fields.

1.5 Division and Modulo with (Pseudo) Mersenne Primes

We now describe a new algorithm for truncated division with Mersenne primes, and more generalized numbers on the form $2^b - c$. We show this implies a fast branch-free computation of $\bmod p$ for Mersenne primes $p = 2^b - 1$. An annoyance in Algorithm 1 is that the if-statement at the end can be slow in case of branch mis-predictions. This method solves that issue.

More specifically, in Algorithm 1, after the last multiplication, we have a number $y < p^2$ and we want to compute the final hash value $y \bmod p$. We obtained this using the following statements, each of which preserves the value modulo p, starting from $y < p^2$:

$y \leftarrow (y \,\&\, p) + (y \gg b)$ ▷ $y < 2p$
if $y \geq p$ **then**
  $y \leftarrow y - p$ ▷ $y < p$

To avoid the if-statement, in Algorithm 4, we suggest a branch-free code that starting from $v < 2^{2b}$ computes both $y = v \bmod p$ and $z = \lfloor v/p \rfloor$ using a small number of AC^0 instructions.

Algorithm 4 For Mersenne prime $p = 2^b - 1$ and $v < 2^{2b}$, compute $y = v \bmod p$ and $z = \lfloor v/p \rfloor$

▷ First we compute $z = \lfloor v/p \rfloor$
$v' = v + 1$
$z \leftarrow ((v' \gg b) + v') \gg b$
▷ Next we compute $y = v \bmod p$ given $z = \lfloor v/p \rfloor$
$y \leftarrow (v + z) \,\&\, p$

In Algorithm 4, we use $z = \lfloor v/p \rfloor$ to compute $y = v \bmod p$. If we only want the division $z = \lfloor v/p \rfloor$, then we can skip the last statement.

Below we will generalize Algorithm 4 to work for arbitrary v, not only $v < 2^{2b}$. Moreover, we will generalize to work for different kinds of primes generalizing Mersenne primes:

Pseudo-Mersenne Primes are primes of the form $2^b - c$, where is usually required that $c < 2^{\lfloor b/2 \rfloor}$ [VTJ14]. Crandal patented a method for working with Pseudo-Mersenne Primes in 1992 [Cra92], why those primes are also sometimes called "Crandal-primes". The method was formalized and extended by Jaewook Chung and Anwar Hasan in 2003 [CH03]. The method we present is simpler with stronger guarantees and better practical performance. We provide a comparison with the Crandal-Chung-Hansan method in Section 4.

Generalized Mersenne Primes also sometimes known as Solinas primes [Sol11], are sparse numbers, that is $f(2^b)$ where $f(x)$ is a low-degree polynomial. Examples from the Internet Research Task Force's document "Elliptic Curves for Security" [AL16]: $p_{25519} = 2^{255} - 19$ and $p_{448} = 2^{448} - 2^{224} - 1$. We simply note that Solinas primes form a special case of Pseudo-Mersenne Primes, where multiplication with c can be done using a few shifts and additions.

We will now first generalize the division from Algorithm 4 to cover arbitrary v and division with an arbitrary Pseudo-Mersenne primes $p = 2^b - c$. This is done in Algorithm 5 below which works also if $p = 2^b - c$ is not a prime. The simple division in Algorithm 4 corresponds to the case where $c = 1$ and $m = 2$.

Algorithm 5 Given integers $p = 2^b - c$ and m. For any $v < (2^b/c)^m$, compute $z = \lfloor v/p \rfloor$

$v' \leftarrow v + c$
$z \leftarrow v' \gg b$
for $m - 1$ times **do**
  $z \leftarrow (z * c + v') \gg b$

The proof that Algorithm 5 correctly computes $z = \lfloor v/p \rfloor$ is provided in Section 4. Note that m can be computed in advance from p, and there is no requirement that it is chosen as

small as possible. For Mersenne and Solinas primes, the multiplication $z * c$ can be done very fast.

Mathematically the algorithm computes the nested division

$$\left\lfloor \frac{v}{q-c} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{\cdots + v + c}{q} \right\rfloor c + v + c}{q} \right\rfloor$$

which is visually similar to the series expansion $\frac{v}{q-c} = \frac{v}{q}\sum_{i=0}^{\infty}(\frac{c}{q})^i = \frac{\frac{\cdots+v}{q}c+v}{q}$. It is natural to truncate this after m steps for a $(c/q)^m$ approximation. The less intuitive part is that we need to add $v+c$ rather than v at each step, to compensate for rounding down the intermediate divisions.

Computing mod We will now compute the mod operation assuming that we have already computed $z = \lfloor v/p \rfloor$. Then

$$v \bmod p = v - pz = v - (2^b - c)z = v - (z \ll b) - c * z, \tag{11}$$

which is only two additions, a shift, and a multiplication with c on top of the division algorithm. As $pz = \lfloor v/p \rfloor p \leq v$ there is no danger of overflow. We can save one operation by noting that if $v = z(2^b - c) + y$, then

$$v \bmod p = y = (v + c * z) \bmod 2^b.$$

This is the method presented in Algorithm 6 and applied with $c = 1$ in Algorithm 4.

Algorithm 6 For integers $p = 2^b - c$ and $z = \lfloor v/p \rfloor$ compute $y = v \bmod p$.

$y \leftarrow (v + z * c)$ & $(2^b - 1)$

Applications to an arbitrary number of buckets In Subsection 1.4 we discussed how $\lfloor \frac{h(x)r}{2^b-1} \rfloor$ provides a most uniform map from $[2^b - 1] \to [r]$. To avoid the division step, we instead considered the map $\lfloor \frac{(h(x)+1)r}{2^b} \rfloor$. However, for primes of the form $2^b - c$, $c > 1$ this approach doesn't provide a most-uniform map. Instead, we may use Algorithm 5 to compute

$$\left\lfloor \frac{h(x)r}{2^b - c} \right\rfloor$$

directly, getting a perfect most-uniform map.

Application to Finger Printing A classical idea by Rabin [Rab81] is to test the equality of two large numbers by comparing their value modulo some random primes (or random irreducible polynomials in a Gallois Field.) A beautiful example of this is King and Sagert's Algorithm for Maintaining the Transitive Closure. [KS02] For such applications we need a reasonably large set of random primes to choose from. The generalized Mersenne primes with c up to $2^{b(1-\varepsilon)}$ are a good candidate set, which from the prime number theorem we expect to contain $\approx 2^{b(1-\varepsilon)}/b$ primes. Each application of algorithm 5 reduces y by a factor $2^{-\varepsilon b}$, so computing the quotient and remainder takes just $m = 1/\varepsilon$ steps.

2 Analysis of second moment estimation using Mersenne primes

In this section, we will prove Theorem 1.2—that a single Mersenne hash function works for Count Sketch. Recall that for each key $x \in [u]$, we have a value $f_x \in \mathbb{Z}$, and the goal was to estimate the second moment $F_2 = \sum_{x \in u} f_x^2$.

We had two functions $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$. For notational convenience, we define $i_x = i(x)$ and $s_x = s(x)$. We let $r = 2^\ell > 1$ and $u > r$ both be powers of two and $p = 2^b - 1 > u$ a Mersenne prime. For each $i \in [r]$, we have a counter $C_i = \sum_{x \in [u]} s_x f_x [i_x = i]$, and we define the estimator $X = \sum_{i \in [r]} C_i^2$. We want to study how well it approximates F_2. We have

$$X = \sum_{i \in [r]} \left(\sum_{x \in [u]} s_x f_x [i_x = i] \right)^2 = \sum_{x,y \in [u]} s_x s_y f_x f_y [i_x = i_y] = \sum_{x \in [u]} f_x^2 + Y, \tag{12}$$

where $Y = \sum_{x,y \in [u], x \neq y} s_x s_y f_x f_y [i_x = i_y]$. The goal is thus to bound mean and variance of the error Y.

As discussed in the introduction, one of the critical steps in the analysis of count sketch in the classical case is eq. (7). We formalize this into the following property:

Property 1 (Sign Cancellation)**.** *For distinct keys $x_0, \dots x_{j-1}$, $j \leq k$ and an expression $A(i_{x_0}, \dots, i_{x_{j-1}}, s_{x_1}, \dots, s_{x_{j-1}})$, which depends on $i_{x_0}, \dots, i_{x_{j-1}}$ and $s_{x_1}, \dots, s_{x_{j-1}}$ but not on s_{x_0}*

$$\mathrm{E}[s_{x_0} A(i_{x_0}, \dots, i_{x_{j-1}}, s_{x_1}, \dots, s_{x_{j-1}})] = 0 \,. \tag{13}$$

In the case where we use a Mersenne prime for our hash function we have that h is uniform in $[2^b - 1]$ and not in $[2^b]$, hence Property 1 is not satisfied. Instead, we have eq. (7) which is almost as good, and will replace Property 1 in the analysis for count sketch. We formalize this as follows:

Property 2 (Sign Near Cancellation)**.** *Given k, p and δ, there exists $t \in [r]$ such that for distinct keys $x_0, \dots x_{j-1}$, $j \leq k$ and an expression $A(i_{x_0}, \dots, i_{x_{j-1}}, s_{x_1}, \dots, s_{x_{j-1}})$, which depends on $i_{x_0}, \dots, i_{x_{j-1}}$ and $s_{x_1}, \dots, s_{x_{j-1}}$, but not on s_{x_0},*

$$\mathrm{E}[s_{x_0} A(i_{x_0}, \dots, i_{x_{j-1}}, s_{x_1}, \dots, s_{x_{j-1}})] = \frac{1}{p} \mathrm{E}[A(i_{x_0}, \dots, i_{x_{j-1}}, s_{x_1}, \dots, s_{x_{j-1}}) \mid i_{x_0} = t]. \tag{14}$$

$$\textit{and} \quad \Pr[i_x = t] \leq (1 + \delta)/r \quad \textit{for any key } x. \tag{15}$$

When the hash function h is not uniform then it is not guaranteed that the collision probability is $1/r$, but (4) showed that for Mersenne primes the collision probability is $(1 + (r-1)/p^2)/r$. We formalize this into the following property.

Property 3 (Low Collisions)**.** *We say the hash function has $(1+\varepsilon)/r$-low collision probability, if for distinct keys $x \neq y$,*

$$\Pr[i_x = i_y] \leq (1 + \varepsilon)/r \,. \tag{16}$$

2.1 The analysis in the classical case

First, as a warm-up for later comparison, we analyse the case where we have Sign Cancellation, but the collision probability bound is only $(1+\varepsilon)/r$. This will come in useful in Section 3 where we will consider the case of an arbitrary number of buckets, not necessarily a power of two.

Lemma 2.1. *If the hash function has Sign Cancellation for $k = 4$ and $(1+\varepsilon)/r$-low collision probability, then*

$$\mathrm{E}[X] = F_2 \tag{17}$$

$$\mathrm{Var}[X] \le 2(1+\varepsilon)(F_2^2 - F_4)/r \le 2(1+\varepsilon)F_2^2/r. \tag{18}$$

Proof. Recall the decomposition $X = F_2 + Y$ from eq. (12). We will first show that $\mathrm{E}[Y] = 0$. By Property 1 we have that $\mathrm{E}[s_x s_y f_x f_y [i_x = i_y]] = 0$ for $x \neq y$ and thus $\mathrm{E}[Y] = \sum_{x,y\in[u],x\neq y} \mathrm{E}[s_x s_y f_x f_y [i_x = i_y]] = 0$.

Now we want to bound the variance of X. We note that since $\mathrm{E}[Y] = 0$ and $X = F_2 + Y$

$$\mathrm{Var}[X] = \mathrm{Var}[Y] = \mathrm{E}[Y^2] = \sum_{\substack{x,y,x',y'\in[u]\\ x\neq y, x'\neq y'}} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])].$$

Now we consider one of the terms $\mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])]$. Suppose that one of the keys, say x, is unique, i.e. $x \notin \{y, x', y'\}$. Then the Sign Cancellation Property implies that

$$\mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] = 0.$$

Thus we can now assume that there are no unique keys. Since $x \neq y$ and $x' \neq y'$, we conclude that $(x, y) = (x', y')$ or $(x, y) = (y', x')$. Therefore

$$\begin{aligned}
\mathrm{Var}[X] &= \sum_{\substack{x,y,x',y'\in[u]\\ x\neq y, x'\neq y'}} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\
&= 2 \sum_{\substack{x,y,x',y'\in[u]\\ x\neq y, (x',y')=(x,y)}} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\
&= 2 \sum_{x,y\in[u],x\neq y} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])^2] \\
&= 2 \sum_{x,y\in[u],x\neq y} \mathrm{E}[(f_x^2 f_y^2 [i_x = i_y])] \\
&\le 2 \sum_{x,y\in[u],x\neq y} (f_x^2 f_y^2)(1+\varepsilon)/r \\
&= 2(1+\varepsilon)(F_2^2 - F_4)/r.
\end{aligned}$$

The inequality follows by Property 3. □

2.2 The analysis of two-for-one using Mersenne primes

We will now analyse the case where the functions $s : [u] \to \{-1, 1\}$ and $i : [u] \to [2^l]$ are constructed as in Algorithm 1 from a single k-universal hash function $h : [u] \to [2^b - 1]$ where $2^b - 1$ is a Mersenne prime. We now only have Sign Near Cancellation. We will show that this does not change the expectation and variance too much. Similarly, to the analysis of the classical case, we will analyse a slightly more general problem, which will be useful in Section 3.

Lemma 2.2. *If we have Sign Near Cancellation with $\Pr[i_x = t] \le (1+\delta)/r$ and $(1+\varepsilon)/r$-low collision probability, then*

$$\mathrm{E}[X] = F_2 + (F_1^2 - F_2)/p^2 \tag{19}$$

$$|\mathrm{E}[X] - F_2| \le F_2(n-1)/p^2 \tag{20}$$

$$\mathrm{Var}[X] \le 2F_2^2/r + F_2^2(2\varepsilon/r + 4(1+\delta)n/(rp^2) + n^2/p^4 - 2/(rn)) \tag{21}$$

Proof. We first bound $\mathrm{E}[s_x s_y f_x f_y [i_x = i_y]]$ for distinct keys $x \neq y$. Let t be the special index given by Sign Near Independence. Using eq. (14) twice we get that

$$\begin{aligned}\mathrm{E}[s_x s_y f_x f_y [i_x = i_y]] &= \mathrm{E}[s_x f_x f_y [i_x = i_y] \mid i_y = t]/p \\ &= \mathrm{E}[s_x f_x f_y [i_x = t]]/p \\ &= \mathrm{E}[f_x f_y [i_x = t] \mid i_x = t]/p \\ &= f_x f_y / p^2 \,.\end{aligned} \tag{22}$$

From this, we can calculate $\mathrm{E}[X]$.

$$\mathrm{E}[X] = F_2 + \sum_{x \neq y} \mathrm{E}[s_x s_y f_x f_y [i_x = i_y]] = F_2 + (F_1^2 - F_2)/p^2.$$

Now we note that $0 \leq F_1^2 \leq n F_2$ by Cauchy-Schwarz, hence we get that $|\,\mathrm{E}[X] - F_2| \leq (n-1)/p^2$.

The same method is applied to the analysis of the variance, which is

$$\mathrm{Var}[X] = \mathrm{Var}[Y] \leq \mathrm{E}[Y^2] = \sum_{x,y,x',y' \in [u], x \neq y, x' \neq y'} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \,.$$

Consider any term in the sum. Suppose some key, say x, is unique in the sense that $x \notin \{y, x', y'\}$. Then we can apply eq. (14). Given that $x \neq y$ and $x' \neq y'$, we have either 2 or 4 such unique keys. If all 4 keys are distinct, as in eq. (22), we get

$$\begin{aligned}&\mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\ &\qquad = \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])]\,\mathrm{E}[s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\ &\qquad = (f_x f_y / p^2)(f_{x'} f_{y'} / p^2) \\ &\qquad = f_x f_y f_{x'} f_{y'} / p^4 \,.\end{aligned}$$

The expected sum over all such terms is thus bounded as

$$\begin{aligned}&\sum_{\text{distinct } x,y,x',y' \in [u]} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\ &\qquad = \sum_{\text{distinct } x,y,x',y' \in [u]} f_x f_y f_{x'} f_{y'} / p^4 \\ &\qquad \leq F_1^4 / p^4 \\ &\qquad \leq F_2^2 n^2 / p^4.\end{aligned} \tag{23}$$

Where the last inequality used Cauchy-Schwarz. We also have to consider all the cases with two unique keys, e.g., x and x' unique while $y = y'$. Then using eq. (14) and eq. (15), we get

$$\begin{aligned}&\mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\ &\qquad = f_x f_{x'} f_y^2 \,\mathrm{E}[s_x s_{x'} [i_x = i_{x'} = i_y]] \\ &\qquad = f_x f_{x'} f_y^2 \,\mathrm{E}[s_{x'} [t = i_{x'} = i_y]]/p \\ &\qquad = f_x f_{x'} f_y^2 \,\mathrm{E}[t = i_y]/p^2 \\ &\qquad \leq f_x f_{x'} f_y^2 (1+\delta)/(rp^2).\end{aligned}$$

Summing over all terms with x and x' unique while $y = y'$, and using Cauchy-Schwarz and $u \leq p$, we get

$$\sum_{\text{distinct } x,x',y} f_x f_{x'} f_y^2 (1+\delta)/(rp^2) \leq F_1^2 F_2 (1+\delta)/(rp^2) \leq F_2^2 n (1+\delta)/(rp^2).$$

There are four ways we can pick the two unique keys $a \in \{x, y\}$ and $b \in \{x', y'\}$, so we conclude that

$$\sum_{\substack{x,y,x',y' \in [u], x \neq y, x' \neq y', \\ (x,y)=(x',y') \vee (x,y)=(y',x')}} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \leq 4F_2^2 n(1+\delta)/(rp^2). \quad (24)$$

Finally, we need to reconsider the terms with two pairs, that is where $(x, y) = (x', y')$ or $(x, y) = (y', x')$. In this case, $(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}]) = f_x^2 f_y^2 [i_x = i_y]$. By eq. (16), we get

$$\begin{aligned} &\sum_{\substack{x,y,x',y' \in [u], x \neq y, x' \neq y', \\ (x,y)=(x',y') \vee (x,y)=(y',x')}} \mathrm{E}[(s_x s_y f_x f_y [i_x = i_y])(s_{x'} s_{y'} f_{x'} f_{y'} [i_{x'} = i_{y'}])] \\ &\qquad = 2 \sum_{x,y \in [u], x \neq y} f_x^2 f_y^2 \Pr[i_x = i_y] \\ &\qquad = 2 \sum_{x,y \in [u], x \neq y} f_x^2 f_y^2 (1+\varepsilon)/r \\ &\qquad = 2(F_2^2 - F_4)(1+\varepsilon)/r. \end{aligned} \quad (25)$$

Adding up add (23), (24), and (25), we get

$$\begin{aligned} \mathrm{Var}[Y] &\leq 2(1+\varepsilon)(F_2^2 - F_4)/r + F_2^2(4(1+\delta)n/(rp^2) + n^2/p^4) \\ &\leq 2F_2^2/r + F_2^2(2\varepsilon/r + 4(1+\delta)n/(rp^2) + n^2/p^4 - 2/(rn)). \end{aligned}$$

This finishes the proof. □

We are now ready to prove Theorem 1.2.

Theorem 1.2. *Let $r > 1$ and $u > r$ be powers of two and let $p = 2^b - 1 > u$ be a Mersenne prime. Suppose we have a 4-universal hash function $h : [u] \to [2^b - 1]$, e.g., generated using Algorithm 1. Suppose $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$ are constructed from h as described in Algorithm 3. Using this i and s in the Count Sketch Algorithm 2, the second moment estimate $X = \sum_{i \in [k]} C_i^2$ satisfies:*

$$\mathrm{E}[X] = F_2 + (F_1^2 - F_2)/p^2, \quad (26)$$

$$|\mathrm{E}[X] - F_2| \leq F_2(n-1)/p^2, \quad (27)$$

$$\mathrm{Var}[X] < 2F_2^2/r. \quad (28)$$

From Equation (9) and Equation (3) we have Sign Near Cancellation with $\Pr[i_x = 2^b - 1] \leq (1 - (r-1)/p)/r$ and Equation (4) $(1 + (r-1)/p^2)/r$-low collision probability. Now Lemma 2.2 give us (26) and (27). Furthermore, we have that

$$\begin{aligned} \mathrm{Var}[X] &\leq 2F_2^2/r + F_2^2(2\varepsilon/r + 4(1+\delta)n/(rp^2) + n^2/p^4 - 2/(rn)) \\ &= 2F_2^2/r + F_2^2(2/p^2 + 4n/(rp^2) + n^2/p^4 - 2/(rn)). \end{aligned}$$

We know that $2 \leq r \leq u/2 \leq (p+1)/4$ and $n \leq u$. This implies that $p \geq 7$ and that $n/p \leq u/p \leq 4/7$. We want to prove that $2/p^2 + 4n/(rp^2) + n^2/p^4 - 2/(rn) \leq 0$ which would prove our result. We get that

$$2/p^2 + 4n/(rp^2) + n^2/p^4 - 2/(rn) \leq 2/p^2 + 4u/(rp^2) + u^2/p^4 - 2/(ru).$$

Now we note that $4u/(rp^2) - 2/(ru) = (2u^2 - p^2)/(up^2 r) \leq 0$ since $u \leq (p+1)/2$ so it maximized when $r = u/2$. We then get that

$$2/p^2 + 4u/(rp^2) + u^2/p^4 - 2/(ru) \leq 2/p^2 + 8/p^2 + u^2/p^4 - 4/u^2.$$

We now use that $u/p \leq (4/7)^2$ and get that

$$2/p^2 + 8/p^2 + u^2/p^4 - 4/u^2 \leq (10 + (4/7)^2 - 4(7/4)^2)/p^2 \leq 0.$$

This finishes the proof of (28) and thus also of Theorem 1.2.

3 Algorithms and analysis with an arbitrary number of buckets

Picture a hash function as throwing balls (keys) into buckets (hash values). In the previous sections we have considered the case of a prime number of buckets. We now consider the case of an arbitrary, not necessarily prime, number of buckets.

We will analyse the collision probability with the most uniform maps introduced in Section 1.4, and later we will show how they can be used in connection with the two-for-one hashing from Section 1.3.3.

3.1 An arbitrary number of buckets

We have a hash function $h : U \to Q$, but we want hash values in R, so we need a map $\mu : Q \to R$, and then use $\mu \circ h$ as our hash function from U to R. We normally assume that the hash values with h are pairwise independent, that is, for any distinct x and y, the hash values $h(x)$ and $h(y)$ are independent, but then $\mu(h(x))$ and $\mu(h(y))$ are also independent. This means that the collision probability can be calculated as

$$\Pr[\mu(h(x)) = \mu(h(y))] = \sum_{i \in R} \Pr[\mu(h(x)) = \mu(h(y)) = i] = \sum_{i \in R} \Pr[\mu(h(x) = i)]^2.$$

This sum of squared probabilities attains is minimum value $1/|R|$ exactly when $\mu(h(x))$ is uniform in R.

Let $q = |Q|$ and $r = |R|$. Suppose that h is 2-universal. Then $h(x)$ is uniform in Q, and then we get the lowest collision probability with $\mu \circ h$ if μ is most uniform as defined in Section 1.4, that is, the number of elements from Q mapping to any $i \in [r]$ is either $\lfloor q/r \rfloor$ or $\lceil q/r \rceil$. To calculate the collision probability, let $a \in [r]$ be such that r divides $q + a$. Then the map μ maps $\lceil q/r \rceil = (q+a)/r$ balls to $r - a$ buckets and $\lfloor q/r \rfloor = (q + a - r)/r$ balls to a buckets. For a key $x \in [u]$, we thus have $r - a$ buckets hit with probability $(1 + a/q)/r$ and a buckets hit with probability $(1 - (r-a)/q)/r$. The collision probability is then

$$\begin{aligned} \Pr[\mu(h(x)) = \mu(h(y))] &= (r-a)((1+a/q)/r)^2 + a((1-(r-a)r/q)/r)^2 \\ &= (1 + a(r-a)/q^2)/r \\ &\leq \left(1 + (r/(2q))^2\right)/r. \end{aligned} \tag{29}$$

Note that the above calculation generalizes the one for (4) which had $a = 1$. We will think of $(r/(2q))^2$ as the general relative rounding cost when we do not have any information about how r divides q.

3.2 Two-for-one hashing from uniform bits to an arbitrary number of buckets

We will now briefly discuss how we get the two-for-one hash functions in count sketches with an arbitrary number r of buckets based on a single 4-universal hash function $h : [u] \to [2^b]$. We want to construct the two hash functions $s : [u] \to \{-1, 1\}$ and $i : [u] \to [r]$. As usual the results with uniform b-bit strings will set the bar that we later compare with when from h we get hash values that are only uniform in $[2^b - 1]$.

The construction of s and i is presented in Algorithm 7.

Algorithm 7 For key $x \in [u]$, compute $i(x) = i_x \in [r]$ and $s(x) = s_x \in \{-1, 1\}$. Uses 4-universal $h : [u] \to [2^b]$.

$h_x \leftarrow h(x)$	$\triangleright$ h_x has b uniform bits
$j_x \leftarrow h_x \,\&\, (2^{b-1} - 1)$	$\triangleright$ j_x gets $b-1$ least significant bits of h_x
$i_x \leftarrow (r * j_x) \gg (b-1)$	$\triangleright$ i_x is most uniform in $[r]$
$a_x \leftarrow h_x \gg (b-1)$	$\triangleright$ a_x gets the most significant bit of h_x
$s_x \leftarrow (a_x \ll 1) - 1$	$\triangleright$ s_x is uniform in $\{-1, 1\}$ and independent of i_x.

The difference relative to Algorithm 3 is the computation of i_x where we now first pick out the $(b-1)$-bit string j_x from h_x, and then apply the most uniform map $(rj_x) \gg (b-1)$ to get i_x. This does not affect s_x which remains independent of i_x, hence we still have Sign Cancellation. But i_x is no longer uniform in $[r]$ and only most uniform so by (29) we have $(1 + (r/2^b)^2)/r$-low collision probability. Now Lemma 2.1 give us $\mathrm{E}[X] = F_2$ and

$$\mathrm{Var}[X] \leq 2(F_2^2 - F_4)\left(1 + (r/2^b)^2\right)/r \leq 2F_2^2\left(1 + (r/2^b)^2\right)/r. \tag{30}$$

3.3 Two-for-one hashing from Mersenne primes to an arbitrary number of buckets

We will now show how to get the two-for-one hash functions in count sketches with an arbitrary number r of buckets based on a single 4-universal hash function $h : [u] \to [2^b - 1]$. Again we want to construct the two hash functions $s : [u] \to \{-1, 1\}$ and $i : [u] \to [r]$. The construction will be the same as we had in Algorithm 7 when h returned uniform values in $[2^b]$ with the change that we set $h_x \leftarrow h(x) + 1$, so that it becomes uniform in $[2^b] \setminus \{0\}$. It is also convenient to swap the sign of the sign-bit s_x setting $s_x \leftarrow 2a_x - 1$ instead of $s_x \leftarrow 1 - 2a_x$. The basic reason is that this makes the analysis cleaner. The resulting algorithm is presented as Algorithm 8.

Algorithm 8 For key $x \in [u]$, compute $i(x) = i_x \in [r]$ and $s(x) = s_x \in \{-1, 1\}$. Uses 4-universal $h : [u] \to [p]$ for Mersenne prime $p = 2^b - 1 \geq u$.

$h_x \leftarrow h(x) + 1$	$\triangleright$ h_x uses b bits uniformly except $h_x \neq 0$
$j_x \leftarrow h_x \,\&\, (2^{b-1} - 1)$	$\triangleright$ j_x gets $b-1$ least significant bits of h_x
$i_x \leftarrow (r * j_x) \gg (b-1)$	$\triangleright$ i_x is quite uniform in $[r]$
$a_x \leftarrow h_x \gg (b-1)$	$\triangleright$ a_x gets the most significant bit of h_x
$s_x \leftarrow 1 - (a_x \ll 1)$	$\triangleright$ s_x is quite uniform in $\{-1, 1\}$ and quite independent of i_x.

The rest of Algorithm 8 is exactly like Algorithm 7, and we will now discuss the new distributions of the resulting variables. We had h_x uniform in $[2^b] \setminus \{0\}$, and then we set $j_x \leftarrow h_x \,\&\, (2^{b-1} - 1)$. Then $j_x \in [2^{b-1}]$ with $\Pr[j_x = 0] = 1/(2^b - 1)$ while $\Pr[j_x = j] = 2/(2^b - 1)$ for all $j > 0$.

Next we set $i_x \leftarrow (rj_x) \gg b-1$. We know from Lemma 1.3 (i) that this is a most uniform map from $[2^{b-1}]$ to $[r]$. It maps a maximal number of elements from $[2^{b-1}]$ to 0, including 0 which had half probability for j_x. We conclude

$$\Pr[i_x = 0] = (\lceil 2^{b-1}/r \rceil 2 - 1)/(2^b - 1) \tag{31}$$

$$\Pr[i_x = i] \in \{\lfloor 2^{b-1}/r \rfloor 2/(2^b-1), \lceil 2^{b-1}/r \rceil 2/(2^b-1)\} \text{ for } i \neq 0. \tag{32}$$

We note that the probability for 0 is in the middle of the two other bounds and often this yields a more uniform distribution on $[r]$ than the most uniform distribution we could get from the uniform distribution on $[2^{b-1}]$.

With more careful calculations, we can get some nicer bounds that we shall later use.

Lemma 3.1. *For any distinct $x, y \in [u]$,*

$$\Pr[i_x = 0] \leq (1 + r/2^b)/r \tag{33}$$

$$\Pr[i_x = i_y] \leq \left(1 + (r/2^b)^2\right)/r. \tag{34}$$

Proof. The proof of (33) is a simple calculation. Using (31) and the fact $\lceil 2^{b-1}/r \rceil \leq (2^{b-1} + r - 1)/r$ we have

$$\begin{aligned}\Pr[i_x = 0] &\leq (2(2^{b-1} + r - 1)/r) - 1)/(2^b - 1) \\ &= \left(1 + (r-1)/(2^b-1)\right)/r \\ &\leq \left(1 + r/2^b\right)/r.\end{aligned}$$

The last inequality follows because $r < u < 2^b$.

For 34, let $q = 2^{b-1}$ and $p = 1/(2q-1)$. We define $a \geq 0$ to be the smallest integer, such that[4] $r \setminus q + a$. In particular this means $\lceil q/r \rceil = (q+a)/r$ and $\lfloor q/r \rfloor = (q - r + a)/r$.

We bound the sum

$$\Pr[i_x = i_y] = \sum_{k=0}^{r-1} \Pr[i_x = k]^2$$

by splitting into three cases: 1) The case $i_x = 0$, where $\Pr[i_x = 0] = (2\lceil q/r \rceil - 1)p$, 2) the $r - a - 1$ indices j where $\Pr[i_x = j] = 2\lceil q/r \rceil p$, and 3) the a indices j st. $\Pr[i_x = j] = 2\lfloor q/r \rfloor p$.

$$\begin{aligned}\Pr[i_x = i_y] &= (2p\lceil q/r \rceil - p)^2 + (r - a - 1)(2p\lceil q/r \rceil)^2 + (r-a)(2p\lfloor q/r \rfloor)^2 \\ &= ((4a+1)r + 4(q+a)(q-a-1))p^2/r \\ &\leq (1 + (r^2 - r)/(2q-1)^2)/r.\end{aligned}$$

The last inequality comes from maximizing over a, which yields $a = (r-1)/2$.

The result now follows from

$$(r^2 - r)/(2q-1)^2 \leq (r - 1/2)^2/(2q-1)^2 \leq (r/(2q))^2, \tag{35}$$

which holds exactly when $r \leq q$.

□

Lemma 3.1 above is all we need to know about the marginal distribution of i_x. However, we also need a replacement for Lemma 1.1 for handling the sign-bit s_x.

[4] Like Knuth we let our divisibility symbol lean leftward.

Lemma 3.2. *Consider distinct keys $x_0, \ldots, x_{j-1}$, $j \leq k$ and an expression $B = s_{x_0} A$ where A depends on $i_{x_0}, \ldots, i_{x_{j-1}}$ and $s_{x_1}, \ldots, s_{x_{j-1}}$ but not s_{x_0}. Then*

$$\mathrm{E}[s_x A] = \mathrm{E}[A \mid i_x = 0]/p. \tag{36}$$

Proof. The proof follows the same idea as that for Lemma 1.1. First we have

$$\mathrm{E}[B] = \mathop{\mathrm{E}}_{h(x_0) \sim \mathcal{U}([2^b] \setminus \{0\})}[B] = \mathop{\mathrm{E}}_{h(x_0) \sim \mathcal{U}[2^b]}[B] 2^b/p - \mathrm{E}[B \mid h(x_0) = 0]/p.$$

With $h(x_0) \sim \mathcal{U}[2^b]$, the bit a_{x_0} is uniform and independent of j_{x_0}, so $s_{x_1} \in \{-1, 1\}$ is uniform and independent of i_{x_0}, and therefore

$$\mathop{\mathrm{E}}_{h(x_0) \sim \mathcal{U}[2^b]}[s_{x_0} A] = 0.$$

Moreover, $h(x_0) = 0$ implies $j_x = x_0$, $i_{x_0} = 0$, $a_{x_0} = 0$, and $s_{x_0} = -1$, so

$$\mathrm{E}[s_{x_0} A] = -\mathrm{E}[s_{x_0} A \mid h(x_0) = 0]/p = \mathrm{E}[A \mid i_{x_0} = 0].$$

□

From Lemma 3.2 and (33) we have Sign Near Cancellation with $\Pr[i_x = 0] \leq (1 + r/2^b)/r$, and (34) implies that we have $(1 + (r/2^b)^2)/r$-low collision probability. We can then use Lemma 2.2 to prove the following result.

Theorem 3.3. *Let u be a power of two, $1 < r \leq u/2$, and let $p = 2^b - 1 > u$ be a Mersenne prime. Suppose we have a 4-universal hash function $h : [u] \to [2^b - 1]$, e.g., generated using Algorithm 1. Suppose $i : [u] \to [r]$ and $s : [u] \to \{-1, 1\}$ are constructed from h as described in Algorithm 8. Using this i and s in the Count Sketch Algorithm 2, the second moment estimate $X = \sum_{i \in [k]} C_i^2$ satisfies:*

$$\mathrm{E}[X] = F_2 + (F_1^2 - F_2)/p^2, \tag{37}$$

$$|\mathrm{E}[X] - F_2| \leq F_2(n-1)/p^2, \tag{38}$$

$$\mathrm{Var}[X] < 2(1 + (r/2^b)^2) F_2^2/r. \tag{39}$$

Now Lemma 2.2 gives us (37) and (38). Furthermore, we have that

$$\begin{aligned}
\mathrm{Var}[X] &\leq 2F_2^2/r + F_2^2(2\varepsilon/r + 4(1+\delta)n/(rp^2) + n^2/p^4 - 2/(rn)) \\
&= 2(1 + (r/2^b)^2) F_2^2/r + F_2^2(4(1 + r/2^b)n/(rp^2) + n^2/p^4 - 2/(rn)) \\
&\leq 2(1 + (r/2^b)^2) F_2^2/r + F_2^2(4(1 + r/p)n/(rp^2) + n^2/p^4 - 2/(rn)).
\end{aligned}$$

We know that $2 \leq r \leq u/2 \leq (p+1)/4$ and $n \leq u$. This implies that $p \geq 7$ and that $n/p \leq u/p \leq 4/7$. If we can prove that $4(1 + r/p)n/(rp^2) + n^2/p^4 - 2/(rn) \leq 0$ then we have the result. We have that

$$\begin{aligned}
4(1 + r/p)n/(rp^2) + n^2/p^4 - 2/(rn) &= 4n/(rp^2) + 4n/(p^3) + n^2/p^4 - 2/(rn) \\
&\leq 4u/(rp^2) + 4u/(p^3) + u^2/p^4 - 2/(ru).
\end{aligned}$$

Again we note that $4u/(rp^2) - 2/(ru) = (2u^2 - p^2)/(up^2 r) \leq 0$ since $u \leq (p+1)/2$ so it maximized when $r = u/2$. We then get that

$$4u/(rp^2) + 4u/(p^3) + u^2/p^4 - 2/(ru) \leq 8/p^2 + 4u/(p^3) + u^2/p^4 - 4/u^2.$$

We now use that $u/p \leq (4/7)^2$ and get that

$$8/p^2 + 4u/(p^3) + u^2/p^4 - 4/u^2. \leq (8 + 4(4/7) + (4/7)^2 - 4(7/4)^2)/p^2 \leq 0.$$

This finishes the proof of (39) and thus also of Theorem 3.3.

4 Quotient and Remainder with Generalized Mersenne Primes

The purpose of this section is to prove the correctness of Algorithm 5. In particular we will prove the following equivalent mathematical statement:

Theorem 4.1. *Given integers $q > c > 0$, $n \geq 1$ and a value x such that*

$$0 \leq x \leq \begin{cases} c(q/c)^n - c & \text{if } c \backslash q \quad (c \text{ divides } q) \\ (q/c)^{n-1}(q-c) & \text{otherwise} \end{cases}.$$

Define the sequence $(v_i)_{i \in [n+1]}$ by $v_0 = 0$ and $v_{i+1} = \left\lfloor \frac{(v_i+1)c+x}{q} \right\rfloor$. Then

$$\left\lfloor \frac{x}{q-c} \right\rfloor = v_n.$$

We note that when $c < q-1$ a sufficient requirement is that $x < (q/c)^n$. For $c = q-1$ we are computing $\lfloor x/1 \rfloor$ so we do not need to run the algorithm at all.

To be more specific, the error $E_i = \lfloor \frac{x}{q-c} \rfloor - v_i$ at each step is bounded by $0 \leq E_i \leq u_{n-i}$, where u_i is a sequence defined by $u_0 = 0$ and $u_{i+1} = \lfloor \frac{q}{c} u_i + 1 \rfloor$. For example, this means that if we stop the algorithm after $n-1$ steps, the error will be at most $u_1 = 1$.

Proof. Write $x = m(q-c) + h$ for non-negative integers m and h with $h < q - c$. Then we get

$$\left\lfloor \frac{x}{q-c} \right\rfloor = m.$$

Let $u_0 = 0$, $u_{i+1} = \lfloor \frac{q}{c} u_i + 1 \rfloor$. By induction $u_i \geq (q/c)^{i-1}$ for $i > 0$. This is trivial for $i = 1$ and $u_{i+1} = \lfloor \frac{q}{c} u_i + 1 \rfloor \geq \lfloor (q/c)^i + 1 \rfloor \geq (q/c)^i$.

Now define $E_i \in \mathbb{Z}$ such that $v_i = m - E_i$. We will show by induction that $0 \leq E_i \leq u_{n-i}$ for $0 \leq i \leq n$ such that $E_n = 0$, which gives the theorem. For a start $E_0 = m \geq 0$ and $E_0 = \lfloor x/(q-c) \rfloor \leq (q/c)^{n-1} \leq u_n$.

For $c \backslash q$ we can be slightly more specific, and support $x \leq c(q/c)^n - c$. This follows by noting that $u_i = \frac{(q/c)^i - 1}{q/c - 1}$ for $i > 0$, since all the q/c terms are integral. Thus for $E_0 = \lfloor x/(q-c) \rfloor \leq u_n$ it suffices to require $x \leq cq^n - c$.

For the induction step we plug in our expressions for x and v_i:

$$\begin{aligned} v_{i+1} &= \left\lfloor \frac{(m - E_i + 1)c + m(q-c) + h}{q} \right\rfloor \\ &= m + \left\lfloor \frac{(-E_i + 1)c + h}{q} \right\rfloor \\ &= m - \left\lceil \frac{(E_i - 1)c - h}{q} \right\rceil. \end{aligned}$$

The lower bound follows easily from $E_i \geq 0$ and $h \leq q - c - 1$:

$$E_{i+1} = \left\lceil \frac{E_i c - h - c}{q} \right\rceil \geq \left\lceil \frac{-q+1}{q} \right\rceil = 0.$$

For the upper bound we use the inductive hypothesis as well as the bound $h \geq 0$:

$$\begin{aligned} E_{i+1} &= \left\lceil \frac{(E_i - 1)c - h}{q} \right\rceil \\ &\leq \left\lceil (u_{n-i} - 1)\frac{c}{q} \right\rceil \\ &= \left\lceil \left\lfloor \frac{q}{c} u_{n-i-1} \right\rfloor \frac{c}{q} \right\rceil \\ &\leq \lceil u_{n-i-1} \rceil \\ &= u_{n-i-1}. \end{aligned}$$

The last equality comes from u_{n-i-1} being an integer. Having thus bounded the errors, the proof is complete. □

We can also note that if the algorithm is repeated more than n times, the error stays at 0, since $\lceil (u_{n-i} - 1)\frac{c}{q} \rceil = \lceil -\frac{c}{q} \rceil = 0$.

4.1 Related Algorithms

Modulus computation by Generalized Mersenne primes is widely used in the Cryptography community. For example, four of the recommended primes in NIST's document "Recommended Elliptic Curves for Federal Government Use" [NIS99] are Generalized Mersenne, as well as primes used in the Internet Research Task Force's document "Elliptic Curves for Security" [AL16]. Naturally, much work has been done on making computations with those primes fast. Articles like "Simple Power Analysis on Fast Modular Reduction with Generalized Mersenne Prime for Elliptic Curve Cryptosystems" [SS06] give very specific algorithms *for each* of many well known such primes. An example is shown in Algorithm 9.

Algorithm 9 Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$

input $c \leftarrow (c_5, c_4, c_3, c_2, c_1, c_0)$, where each c_i is a 64-bit word, and $0 \leq c < p_{192}^2$.
$s_0 \leftarrow (c_2, c_1, c_0)$
$s_1 \leftarrow (0, c_3, c_3)$
$s_2 \leftarrow (c_4, c_4, 0)$
$s_3 \leftarrow (c_5, c_5, c_5)$
return $s_0 + s_1 + s_2 + s_3 \mod p_{192}$.

Division by Mersenne primes is a less common task, but a number of well known division algorithms can be specialized, such as classical trial division, Montgomery's method and Barrett reduction.

The state of the art appears to be the modified Crandall Algorithm by Chung and Hasan [CH06]. This algorithm, given in Algorithm 10 modifies Crandall's algorithm [Cra92] from 1992 to compute division as well as modulo for generalized $2^b - c$ Mersenne primes.[5]

[5] Chung and Hasan also have an earlier, simpler algorithm from 2003 [CH03], but it appears to give the wrong result for many simple cases. This appears in part to be because of a lack of the "clean up" while-loop at the end of Algorithm 10.

Algorithm 10 Crandall, Chung, Hassan algorithm. For $p = 2^b - c$, computes q, r such that $x = qp + r$ and $r < p$.

$q_0 \leftarrow x \gg n$
$r_0 \leftarrow x \,\&\, (2^b - 1)$
$q \leftarrow q_0, r \leftarrow r_0$
$i \leftarrow 0$
while $q_i > 0$ **do**
    $t \leftarrow q_i * c$
    $q_{i+1} \leftarrow t \gg n$
    $r_{i+1} \leftarrow t \,\&\, (2^b - 1)$
    $q \leftarrow q + q_{i+1}$
    $r \leftarrow r + r_{i+1}$
    $i \leftarrow i + 1$
$t \leftarrow 2^b - c$
while $r \geq t$ **do**
    $r \leftarrow r - t$
    $q \leftarrow q + 1$

The authors state that for $2n + \ell$ bit input, Algorithm 10 requires at most s iterations of the first loop, if $c < 2^{((s-1)n-\ell)/s}$. This corresponds roughly to the requirement $x < 2^b(2^b/c)^s$, similar to ours. Unfortunately, the algorithm ends up doing double work, by computing the quotient and remainder concurrently. The algorithm also suffers from the extra while loop for "cleaning up" the computations after the main loop. In practice, our method is 2-3 times faster. See Section 5 for an empirical comparison.

5 Experiments

We perform experiments on fast implementations of Mersenne hashing (Algorithm 1) and our Mersenne division algorithm (Algorithm 5). All code is available in our repository github.com/thomasahle/mersenne and compiled with `gcc -O3`.

We tested Algorithm 1 against hashing over the finite fields $GF(2^{64})$ and $GF(2^{32})$. The later is implemented, following Lemire [LK14], using the "Carry-less multiplication' instruction, CLMUL, supported by AMD and Intel processors [GK10].[6] We hash a large number of 64-bit keys into $[p]$ for $p = 2^{89} - 1$ using k-universal hashing for $k \in \{2, 4, 8\}$. Since the intermediate values of our calculations take up to $64 + 89$ bits, all computations of Algorithm 1 are done with 128-bit output registers. We perform the same experiment with $p = 2^{61} - 1$. This allows us to do multiplications without splitting into multiple words, at the cost of a slightly shorter key range.

[6] More precisely, given two b-bit numbers $\alpha = \sum_{i=0}^{b-1} \alpha_i 2^i$ and $\beta = \sum_{i=0}^{b-1} \beta_i 2^i$ the CLMUL instructions calculates $\gamma = \sum_{i=0}^{2b-2} \gamma_i 2^i$, where $\gamma_i = \bigoplus_{j=0}^{j} \alpha_i \beta_{j-i}$. If we view α and β as elements in $GF(2)[x]$ then the CLMUL instruction corresponds to polynomial multiplication. We can then calculate multiplication in a finite field, $GF(2^b)$, efficiently by noting that for any irreducible polynomial $p(x) \in GF(2)[x]$ of degree b then $GF(2^b)$ is isomorphic to $GF(2)[x]/p(x)$. If we choose $p(x)$ such that the degree of $p(x) - 2^b$ is at most $b/2$ then modulus $p(x)$ can be calculated using two CLMUL instructions. For $GF(2^{64})$ we use the polynomial $p(x) = x^{64} + x^4 + x^3 + x + 1$ and for $GF(2^{32})$ we use the polynomial $p(x) = x^{32} + x^7 + x^6 + x^2 + 1$.

k	CPU	Mult-Shift	Mersenne $p = 2^{89} - 1$	Carry-less GF(2^{64})
2	a	**6.4**	23.6	15.1
	b	**7.7**	19.0	16.7
	c	**7.2**	28.3	16.6
4	a	99.3	**65.7**	**65.7**
	b	157.7	**68.7**	**68.8**
	c	117.4	85.2	**67.5**
8	a	1615.8	**178.4**	242.4
	b	1642.3	**187.4**	246.8
	c	1949.9	228.1	**224.1**

k	CPU	Mult-Shift	Mersenne $p = 2^{61} - 1$	Carry-less GF(2^{32})
2	a	**4.4**	13.6	13.0
	b	**3.3**	14.2	13.0
	c	**3.2**	16.5	18.4
4	a	57.6	**31.6**	60.3
	a	54.6	**34.3**	58.8
	c	61.2	**41.9**	74.5
8	a	650.7	**88.0**	218.7
	b	635.6	**88.0**	212.0
	c	750.8	**127.8**	253.2

Table 1: Milliseconds for 10^7 k-universal hashing operations. The standard deviation is less than ±1ms. The three CPUs tested are a) Intel Core i7-8850H; b) Intel Core i7-86650U; c) Intel Xeon E5-2690 .

The results in Table 1 show that our methods outperform carry-less Multiplication for larger k, while being slower for $k = 2$. For $k = 2$ the multiply-shift scheme [Die96] is better yet, so in carry-less multiplication is nearly completely dominated. For $k = 4$, which we use for Count Sketch, the results are a toss-up for $p = 2^{89} - 1$, but the Mersenne primes are much faster for $p = 2^{61} - 1$. We also note that our methods are more portable than carry-less, and we keep the two-for-one advantages described in the article.

We tested Algorithm 5 against the state of the art modified Crandall's algorithm by Chung and Hasan (Algorithm 10), as well as the built-in truncated division algorithm in the GNU MultiPrecision Library, GMP [Gra10].

b	Crandall	Algorithm 5	GMP
32	396	**138**	149
64	381	**142**	161
128	564	**157**	239
256	433	**187**	632
512	687	**291**	1215
1024	885	**358**	2802

b	Crandall	Algorithm 5	GMP
32	497	149	**120**
64	513	198	**191**
128	538	**207**	293
256	571	**222**	603
512	656	**294**	1167
1024	786	**372**	2633

Table 2: Milliseconds for 10^7 divisions of $2b$-bit numbers with $p = 2^b - 1$. The standard deviation is less than ±10ms. On the left, Intel Core i7-8850H. On the right, Intel Xeon E5-2690 v4.

The results in Table 2 show that our method always outperforms the modified Crandall's algorithm, which itself outperforms GMP's division at larger bit-lengths. At shorter bit-lengths it is mostly a toss-up between our method and GMP's.

We note that our code for this experiment is implemented entirely in GMP, which includes some overhead that could perhaps be avoided in an implementation closer to the metal. This overhead is very visible when comparing Table 1 and Table 2, suggesting that an optimized Algorithm 5 would beat GMP even at short bit-lengths.

References

[AL16] S. Turner A. Langley, M. Hamburg. Elliptic Curves for Security. RFC 7748, RFC Editor, January 2016.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999. Announced at STOC'96.

[CCF04] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004. Announced at ICALP'02.

[CH03] Jaewook Chung and Anwar Hasan. More generalized mersenne numbers. In *International Workshop on Selected Areas in Cryptography*, pages 335–347. Springer, 2003.

[CH06] Jaewook Chung and M Anwar Hasan. Low-weight polynomial form integers for efficient modular multiplication. *IEEE Transactions on Computers*, 56(1):44–57, 2006.

[Cra92] Richard E Crandall. Method and apparatus for public key exchange in a cryptographic system, October 27 1992. US Patent 5,159,632.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. Announced at STOC'77.

[Die96] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 569–580, 1996.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. Announced at FOCS'82.

[GK10] Shay Gueron and Michael Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Information Processing Letters*, 110(14):549 – 553, 2010.

[Gra10] Torbjörn Granlund. Gnu multiple precision arithmetic library. *http://gmplib.org/*, 2010.

[KS02] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002.

[LK14] Daniel Lemire and Owen Kaser. Strongly universal string hashing is fast. *The Computer Journal*, 57(11):1624–1638, 2014.

[Moo89] John Moody. Fast learning in multi-resolution hierarchies. In *Advances in neural information processing systems*, pages 29–39, 1989.

[NIS99] NIST. Recommended elliptic curves for federal government use, 1999.

[Rab81] Michael O Rabin. Fingerprinting by random polynomials. *Technical report*, 1981.

[Sie04] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. Comput.*, 33(3):505–543, 2004. Announced at FOCS'89.

[Sol11] Jerome A. Solinas. *Pseudo-Mersenne Prime*, pages 992–992. Springer US, Boston, MA, 2011.

[SS06] Yasuyuki Sakai and Kouichi Sakurai. Simple power analysis on fast modular reduction with generalized mersenne prime for elliptic curve cryptosystems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 89(1):231–237, 2006.

[Tho13] Mikkel Thorup. Simple tabulation, fast expanders, double tabulation, and high independence. In *Proc. 25th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 90–99, 2013.

[TZ12] Mikkel Thorup and Yin Zhang. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing*, 41(2):293–331, 2012.

[VTJ14] Henk CA Van Tilborg and Sushil Jajodia. *Encyclopedia of cryptography and security*. Springer Science & Business Media, 2014.

[WC81] Mark N. Wegman and Larry Carter. New classes and applications of hash functions. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. Announced at FOCS'79.

[WDL+09] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.