The ZPrize PoSW optimization contest is a thorough competition of all aspects, requires a deep understanding of Zero Knowledge proving systems, extensive knowledge of Cryptography and Mathematics, years of experiences of software/hardware co-design targeting high performance. Fortunately our team has always been pursuing those for a long period and we are very happy to share with you our optimization schemes, which could bring new ideas and open the design space for ZKP prover acceleration.

As to PoSW performance, two critical vulnerabilities are found by us. Especially, utilizing these vulnerabilities, the PoSW protocol can be broken and have much higher performance than other solutions.

1 Source Code Repos

- CUDA-wrapper

  https://github.com/Trapdoor-Tech/TrapdoorTech-zprize-crypto-cuda
- CUDA-kernels

  https://github.com/Trapdoor-Tech/TrapdoorTech-zprize-ec-gpu-kernels
- GPU-API-Layer

  https://github.com/Trapdoor-Tech/TrapdoorTech-zprize-ec-gpu-common
- snarkVM

  https://github.com/Trapdoor-Tech/TrapdoorTech-zprize-snarkVM
- Zprize POSW (GPU)

  https://github.com/Trapdoor-Tech/TrapdoorTech-zprize-posw-gpu

You can follow the intructions (Readme) in Zprize POSW (GPU) to compile and run the benchmark.

2 Performance Summary

The performance is measured with hardware configuration provided by CoreWeave GPU cloud platform:

- GPU - 1 NVIDIA RTX A5000
- CPU - 6 cores (AMD EPYC 7413 Processor)
- Memory - 6G memory
- 40G storage space
- Ubuntu 20.04

The TPS result is **56.5** tps or equivalently **1130** proofs generated in 20 seconds. The performance benchmark details are explained in "Performance Benchmark Explanation" section.

3 Critical Vulnerabilities

3.1 Batch size in proof is NOT checked in verification.

The batch_size is **NOT** checked to be consistency with batch_size provided in proof. The verify_batch_prepared function is declared in marlin.rs.

```
    fn verify_batch_prepared<B: std::borrow::Borrow<Self::VerifierInput>>(
        prepared_verifying_key: &<Self::VerifyingKey as Prepare>::Prepared,
        public_inputs: &[B],
        proof: &Self::Proof,
    ) -> Result<bool, SNARKError> {
```

In the function, the batch size is calculated as the array length of "public_inputs".

```
let batch_size = public_inputs.len();
```

But the batch size is NOT checked and required to be same as in provided proof. That's to say, the batch size in proof is NOT checked by verification logic.

By iterating different batch size in one valid proof, more and more proofs can be generated very very fast. And the POSW protocol is cracked.

3.2 POSW Circuit Synthesize and Round 1 calculation can be bypassed

In Round 1, $W(x)$ (public input and witness encoding) , $Z_a(X)$ and $Z_b(x)$ are committed. However, those polynomials' degrees are NOT bounded when committing.

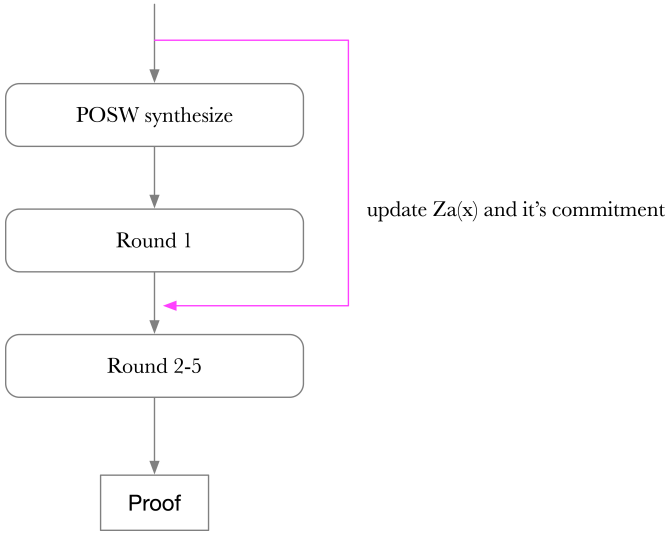
```
    pub fn first_round_polynomial_info(batch_size: usize) -> BTreeMap<PolynomialLabel, PolynomialInfo> {
        let mut polynomials = Vec::new();

        for i in 0..batch_size {
            polynomials.push(PolynomialInfo::new(witness_label("w", i), None, Self::zk_bound()));
            polynomials.push(PolynomialInfo::new(witness_label("z_a", i), None, Self::zk_bound()));
            polynomials.push(PolynomialInfo::new(witness_label("z_b", i), None, Self::zk_bound()));
        }
        if MM::ZK {
            polynomials.push(PolynomialInfo::new("mask_poly".to_string(), None, None));
        }
        polynomials.into_iter().map(|info| (info.label().into(), info)).collect()
    }
```

That's to say, for example, $Z_a(x)$ can be replaced by $Z'_a(x)$:

$$Z'_a(x) = Z_a(x) + rand * V_H(x)$$

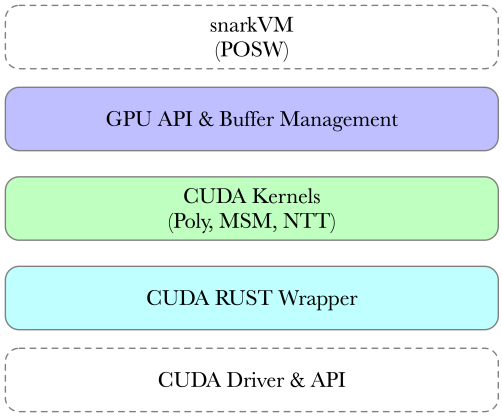
Although the randomness of Marlin protocol is "disabled", due to the fact $Z_a(x)$'s degree is NOT limited, $Z_a(x)$ can be "randomized" by adding vanishing function. The same way can be used on $Z_b(x)$ or $W(x)$ too. As you know, $V_H(x)$'s commitment calculation is easy and can be pre-prepared. It's very easy to update the $Z'_a(x)$'s commitment by adding $V_H(x)$'s commitment times $rand$.

Using the vulnerability, POSW circuit synthesize and Round 1 calculation can be bypassed, illustrated as follows:



4 GPU CUDA Management Stack

For the performance optimization, we'd like to introduce the complete and functional GPU CUDA management stack, that we designed and implemented to accelerate the POSW (marlin part) most computation work done efficiently by GPU.



4.1 CUDA RUST Wrapper

Since we choose GPU as our ZKP hardware acceleration solution, it is very important to have a grounding wrapper for GPU programming. CUDA itself is a versatile GPU development language. In order to fully utilize its convenience, we built a wrapper in Rust to call CUDA functions via Rust FFI. The wrapper itself is not equivalent to the whole CUDA framework, of course, but it contains basic capabilities might be used during GPU programming. Besides, the wrapper is rather light weighted and handy to support new CUDA capabilities.

The following capabilities are included in the Wrapper:

1. CUDA context initializing, destroying, syncing
2. CUDA device information retrieving
3. CUDA kernel finding functions, preparing parameters and launching
4. CUDA memory copying and managing

With these capabilities, we can quickly enable any CUDA device and utilize their cutting edge features.

4.2 CUDA Kernels

To fully utilize GPU device in zero knowledge proving, our CUDA kernel must implement three basic kinds of operators: finite field operators, elliptic curve operators and specialized operators like NTT/MSM. We basically reused some of the fundamental operators that were implemented by Filecoin/SupraNational team, but added another bunch of top level operators according to our demand.

Those top level operators that are implemented by us:

1. Polynomial operators: `add`, `sub`, `mul`, `inv`, `pow`, etc.
2. MSM operators: point addition/mixed addition, Lookup Table algorithm

4.3 GPU API layer / buffer container

It is not enough to have only a wrapper layer for basic CUDA capabilities. In order to conveniently use GPU device in any ZKP systems, we have built an API layer on top of the wrapper. The API layer represents a GPU buffer as a polynomial, allows one to do polynomial arithmetics on it.

Another improvement is GPU buffer container which aims to maximize GPU bandwidth utilization. The container have 4 primitive operations, which are `Ask`, `Recycle`, `Find` and `Save`. Those primitives can nicely meet our demand for GPU buffer life cycle management, saving a lot of time of GPU buffer allocation/revocation/reallocation.

5 FFT optimization

Thanks for the Protocol Lab's contribution. We implemented our GPU-based FFT referring to https://github.com/filecoin-project/ec-gpu/blob/master/ec-gpu-gen/src/cl/fft.cl from filecoin-project ec-gpu project.

And for PoSW, we also implemented the following optimizations:

1. The concrete problem sizes in PoSW, which are almost $2^{15}, 2^{16}, 2^{17}$, are really small, so that we can cache all the related omegas instead of computing them in runtime.
2. Use share memory to store twiddle factors.

6 Multiexp acceleration

6.1 Curve Isogeny / Endomorphism

An important feature of Elliptic Curves is that they are often isomorphic and can be transformed from one curve to another. For example the general curve form is Weierstrass Form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \mod \mathbb{F}_p$$

It can be transformed to Short Weierstrass Form:

$$y^2 = x^3 + ax + b \mod \mathbb{F}_p$$

where characteristic of $\mathbb{F}_p$ is not 2 or 3 . This transformation method is called Curve Isogeny.

The point addition on Short Weierstrass Curves costs 11M+5S according to http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl. Mixed point addition costs 7M+4S according to http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-madd-2007-bl.

Short Weierstrass Curves is not the fastest curve that point addition/mixed addition takes place. Another well known curve form is Twisted Edwards Curves, introduced by Daniel Bernstein et al in 2008 in their paper Twisted Edwards Curves. The curve formula is depicted as below:

$$ax^2 + y^2 = 1 + dx^2y^2 \mod \mathbb{F}_p$$

It has the fastest point addition by far and we can also have unified point addition formula. ZCash team used this trick to generate a Baby JubJub curve on top of BLS12-381. However their purpose is not to accelerate MSM, but to make use of the unified point addition formula and simplify their signature circuit.

The cost of unified addition on Twisted Edwards Curves can be reduced to 8M+1k according to http://www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#addition-add-2008-hwcd-3, while mixed addition costs 7M+1k according to http://www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#addition-madd-2008-hwcd-3, by introducing an extra field element record $t = x * y$ though, as described in Huseyin Hisil et al. paper Twisted Edwards Curves Revisited.

A natural idea of optimizing MSM is to utilize curve isogeny, which means we can transform all the points on BLS12-377 to its isomorphic Twisted Edwards curve, and run MSM on that curve to finalize the result, then transform from Twisted Edwards curve back to BLS12-377. We can have all the awesome features of Twisted Edwards curve, while retaining the general Short Weierstrass curve representation.

Not all Short Weierstrass Curves can be transformed to Twisted Edwards Curves. According to https://en.wikipedia.org/wiki/Montgomery_curve, we know that under certain conditions, a Short Weierstrass Curve is isomorphic to a Montgomery Curve, and a Montgomery Curve is isomorphic to a Twisted Edwards Curve. In order to transform from Short Weierstrass to Montgomery form, We quote:

> In contrast, an elliptic curve over base field $\mathbb{F}$ in Weierstrass form $E_{a,b} : v^2 = t^3 + at + b$ can be converted to Montgomery form if and only if has order divisible by four and satisfies the following conditions:
>
> 1. $z^3 + az + b = 0$ has at least one root $\alpha \in \mathbb{F}$; and
> 2. $3\alpha^2 + a$ is a quadratic residue in $\mathbb{F}$
>
> When these conditions are satisfied, then for $s = (\sqrt{3\alpha^2 + a})^{-1}$ we have the mapping
>
> $\psi^{-1} : E_{a,b} \to M_{A,B}$
>
> $(t, v) \mapsto (x, y) = (s(t - \alpha, sv)), A = 3\alpha s, B = s$

Then from Montgomery to Twisted Edwards, we quote:

> $2\frac{a+d}{a-d} = A$
>
> $\frac{4}{a-d} = B$
>
> $(u, v) \mapsto (u/v, (u - 1)/(u + 1))$

and that's it!

Very luckily, BLS12-377 can be transformed. The Short Weierstrass Form equation of BLS12-377 is:

$$y^2 = ax^3 + b \mod \mathbb{F}_p$$

with

$$a = 0, b = 1, p = 258664426012969094010652733694893533536393512754914660539884262666720468348340822774968888139573360124440$$

We choose

$$A = 305670708996688898721215847896588822742454717287192848948835383955084191963464476825105908353090089367312402257 93$$
$$B = 113327392486723791340039350366245770860783363395096105577549129978801514727083865406602966047408507739599254478215$$

for its Montgomery isogeny $M_{A,B}$, then transform to $E_{a,d}$ with

$$a = 157163064917902313978814213261261898218646390773518349738660969080500653509624033038447657619791437448628296189665$$
$$d = 101501361095066780517536410023107951769097300825221174390295061910482811707540513312796446149590693954692781734188$$

We can even optimize further by making $a = -1$ using its endomorphism. The process is rather simple and can be found on paper, we are not going to demonstrate it here.

So the point addition optimization is rather straight forward here. Firstly, we transform all the base points which are on BLS12-377 to another set of base points on a Twisted Edwards curve that is isomorphic to BLS12-377. Secondly, we do the MSM (mainly operations are point additions/mixed point additions) on it and get an accumulated point. Finally, we transform that single point to BLS12-377 again.

Since MSM is mostly used to commit to a polynomial, those base points are always fixed. This optimization approach should be valid for all ZKP systems, as long as the curve transformation is valid. The performance improvement should be around **30%~40%**.

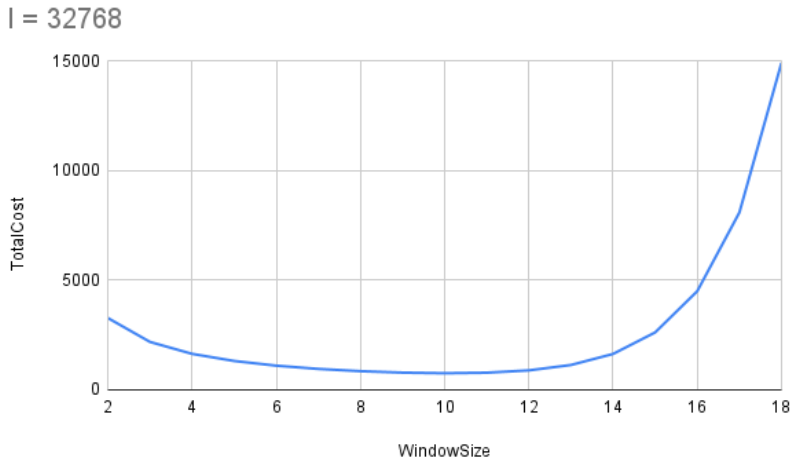
6.2 Pippenger Algorithm V.S. Lookup Table

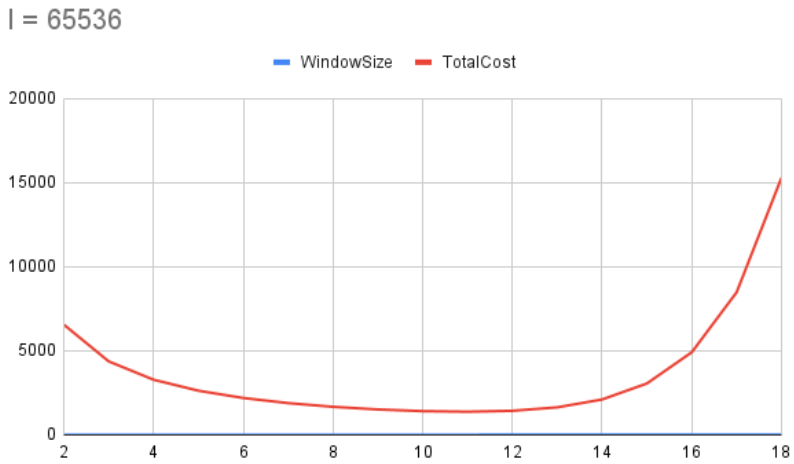
The well known algorithm for Elliptic Curve multi-scalar multiplication is Pippenger's algorithm. It divides scalars into a bunch of windows, for each window it uses buckets to accumulate all the base points, then it scans the windows right-to-left and get the final result. The time complexity for Pippenger's algorithm is comprised of 3 major parts (assuming l the number of scalars, n the number of bits of each scalar, w is the window size, $m = n/w$ the number of windows, $k = 2^w$ the number of buckets per each window):

1. add base points into different buckets, per window. This shall cost $O(l * n/w) = O(l * m)$ in total
2. go through all buckets' results and accumulate, per window. The cost of this part depends on k and w, which is $O(k * n/w) = O(2^w * m)$ in total
3. accumulate all windows' results. This is rather light weighted as we only have to do double-and-add for each window. The cost in total is $O(w * m) = O(n)$

It is easy to see that the total cost depends on the function $O(l, w) = n/w * (l + 2^w) + n = n(\frac{l+2^w}{w} + 1)$. In PoSW we have $n = 256$, $l = 32768, 65536$, depends on which polynomial we are committing to. So the best w can be decided:

For $l = 32768$, best $w = 12$; for $l = 65536$, best $w = 13$. One may further divide scalars into k chunks, each contains l/k scalars. That would be an empirical process and the optimal k depends on one's GPU device. Another consideration is point addition v.s. mixed addition. Usually mixed point addition would be faster than addition, so one should consider the bucket accumulation time bigger than expected $O(2^w * m)$.





Another straight forward MSM algorithm would be the Lookup Table method. First we calculate all the necessary results of $sP_i, s = 1..2^w - 1, i = 1..l$ and store the results in a lookup table, then we do point addition per window according to the scalar in that window. The main cost would be:

1. accumulate all points per window. This would cost $O(l * n/w) = O(l * m)$
2. accumulating all windows' results, which shall cost $O(n)$.

Compared to Pippenger's algorithm, this Lookup Table method can save the bucket accumulation time and should be faster than Pippenger's algorithm. However this method requires huge memory space, as the pre-calculated table would need $O(l * (2^w - 1))$ results. For BLS12-377, which need 96 bytes to store a Short Weierstrass affine point, or 144 bytes to store a Extended Twisted Edwards affine point, a typical $l = 32768$ or $l = 65536$, $w = 12$ would cost 24 or 36 Gigabytes. For those GPU devices which have buffer that are less than 24 or 36 Gigabytes, the trade-off between time and space should be carefully considered.

In our final implementation, the Lookup Table MSM method is used since we can obtain better performance when GPU memory is big enough. Our measurement on an RTX 3090 card shows that Lookup Table method have **50%~100%** performance improvement.

6.3 2-NAF for Window Scanning

Either Pippenger or Lookup Table algorithm would require scanning all the windows of scalars in a right-to-left order. The more bits scanned in one window, the more efficient our MSM algorithm is. A minor improvement would be to use 2-NAF for our Lookup Table algorithm, which is, to use one pre-calculated table $sP_i, s = 1..2^w - 1, i = 1..l$ to represent $[-2^wP_i, \ldots, (2^w - 1)P_i], i = 1..l$, thus increase the window size by 1 bit. This is possible since it's easy to get the negative point $-P$ by reversing y, namely $P = (x, y), -P = (x, -y)$, so no need to store all those $-P$ affine points.

To enable 2-NAF for MSM, we treat the bits inside one window $b_i = (\overbrace{100..1010}^{w+1})_2$ by subtracting a point $2^w P_i$ first, then add them back in the window accumulation stage. Since we only add back at the final stage, we only need to pre-calculate the accumulation result of $P = \sum_{i=1}^{n} 2^w P_i$, which can be computed while generating lookup tables.

W_1	W_0	
Thread_n1+1	Thread_0	
G_0 G_1 G_2	G_0 G_1 G_2	Subtracting [2^w]P_i for each Base Point
Thread_n2	Thread_n1	
G_n G_n+1 G_n+2	G_n G_n+1 G_n+2	Subtracting [2^w]P_i for each Base Point
Accumulate all threads results and add back [2^w]P	Accumulate all threads results and add back [2^w]P	

This 2-NAF method can improve our TPS by around **4%**.

7 Marlin protocol optimization

7.1 Committing to Lagrange Polynomial

There are a lot of protocol level optimizations by Aleo team, mostly focusing on general cases. However the ZPrize puzzle - PoSW circuit proving - is a very special case that we can observe some patterns in the witness or coefficients matrix. One observation is that in the third round of PoSW Marlin prover, when we are calculating the matrix sum of A, B, C, their evaluations on K is limited. Since the PoSW circuit is not very big, the non-zero evaluations on K does not actually have a length of $|NonZeroDomain|$, or $|D_{nz}|$ for short. For example, for matrix A its non-zero evaluations have only 37908 items, which means any other polynomials multiply by A's evaluations on K, namely $eval_A(x)$, also have a maximum of 37908 non-zero items.

Take the polynomial g_A for example. By definition we have:

$g_A(x) = f(x) - f(0)$, while $f(x) = \frac{v_H(\alpha)v_H(\beta)eval_A(x)}{(\beta - row_A(x))(\alpha - col_A(x))}$

If we commit the polynomial by its coefficients form, we could have at most 65536 non-zero items (in fact 65535, as $f(0)$ is subtracted). If we commit it using a set of Lagrange basis, those non-zero items would be at most 37908, saves almost half of the MSM commitment computation time for g_A.

Now the only problem for us is that we don't have the set of Lagrange basis for commitment. What we have is just a set of G_i, which is used for committing polynomials in coefficients form. We need to calculate Lagrange basis by ourselves.

Wikipedia has a good tutorial on how to calculate a set of Lagrange basis. First we choose the roots of unity $\omega^i_{D_{nz}}, i = 0..|D_{nz}| - 1$ on non zero domain to be evaluated, since our polynomials are represented by their evaluations on those roots of unity.

$l_j(x) = \prod_{0 \le m \le k, m \ne j} \frac{x - x_m}{x_j - x_m}, x_i = \omega^i_{D_{nz}}$

Then we can represent a polynomial by its evaluations and the set of Lagrange basis.

$L(x) = \sum_{j=0} y_j l_j(x)$, while y_j is the evaluation of the polynomial at $\omega^j_{D_{nz}}$.

We commit all the $l_j(x)$ using the provided set of G_i, that would be our Lagrange basis for committing polynomials in evaluation form. This trick can help us reduce the commitment time for g_A, g_B, g_C. Typically, committing those three polynomials using their coefficients requires $l = 3 * 65535 = 196605$ length MSM, while committing with Lagrange form only requires $l = 37908 + 38031 + 48434 = 124373$ length MSM, which is only **63%** of previous time.

7.2 t(x) evaluation optimization

In Round 2, t function is expressed as follows:

$$r_M(X, Y) = \sum_{k \in H} r(X, k)\tilde{M}(k, Y)$$

$$t(X) = \sum_{i \in a,b,c} \eta_i r_M(\alpha, X)$$

To get t function, the evaluation of t over H domain is calculated and then coefficients of the t function can be obtained through FFT on H domain. The simplest way to get the evaluation is to scan the 2 dimension of A/B/C matrix. The performance is not good enough if the matrix is sparse. By checking the matrix, the following facts are found:

- The reindex calculation can be done beforehand. Due to the fact, the evaluation is only related with circuit matrix, the reindex mapping can be done beforehand.

- There are a lot of one coefficients in the matrix. If coefficient is one , the mulplication of $r(X,k)\tilde{M}(k,Y)$ can be simplified to $r(X,k)$.
- There are many non-zero coefficients on column 0. The t(0) evaluation can be handled sepratedly.

8 Performance Benchmark Explanation

We do the following tiny modifications to the benchmark, not only to align with our marlin prover optimizations but also **comply with** the constrains.

1. Change the target_difficulty to 1 to make sure no target proof can be generated in 20 seconds.
2. Move difficulty check from benchmark into the marlin prover itself.

   According to the above **3.2**, since the Round 1 can be bypass except the 1st time, we move the difficulty target check into marlin prover. The proof validity is verified outside of the prover loop.
3. Use multiple provers, one prover one CPU thread, instead of original only one prover. And all provers share the same proof counter, so the final numbers of proofs are the sum of each single prover proof numbers. Obviously, this is a more reasonable way to check the miner throughput especially the real PoSW scene is considered. Besides, just similar to the real scene, our harness hardware capability is also fixed, so we should try our best to fully utilize the hardware capability like miners do.
4. Load our precomupte data once **OUTSIDE** of the benchmark time measurement. In fact, as shown above, this "precompute" is our optimization. In practice, all the "precompute" data can be repared offline, and loaded only once when prover is launched. So it is unnecessary to take this duration into account.

References

Marlin Paper

Twisted Edwards Curves

Twisted Edwards Curves Revisited

Lagrange polynomial