12 PlonK

12.1 Number Theoretical Trasnform: Universal Polynomial Accelerator

In the previous sections, when considering Groth16, we have seen the idea that polynomials are powerful encoders of data. To encode a set of N values $a_0, \dots, a_{N-1} \in \mathbb{F}$, we interpolate a polynomial $p(x)$ that at specific points $x_0, \dots, x_{N-1} \in \mathbb{F}$ evaluates to these values. The only condition we impose on these points is that they are distinct (unless, the interpolation would not be properly defined). This way, generally, we have the following interpolation problem:

$$p(x_j) = a_j, \quad j = 0, \dots, N-1$$

Particularly, in Groth16 our choice of points was $x_j = j$ for $j = 0, \dots, N-1$, which, for the large enough finite field $\mathbb{F}$, does not cause any issues. However, the complexity of interpolation in this case is not optimal. Let us see why.

Recall that the interpolation formula (see Section 2 for details) is given by:

$$p(x) = \sum_{i=0}^{N-1} a_i \ell_i(x), \quad \ell_i(x) = \prod_{j=0, j\neq i}^{N-1} \frac{x - x_j}{x_i - x_j}.$$

The naive evaluation of this formula requires $O(N^2)$ operations: we need to compute each ℓ_i, costing $O(N)$ operations, and then sum them up with, again, $O(N)$ operations.

With the specific choice of points $\{x_j\}_{0 \le j < N}$, we can do much better: in fact, we can reduce the complexity to $O(N \log N)$ operations or even $O(N)$. This is done by utilizing several techniques, including the **Barycentric Interpolation** and the N**th roots of unity**.

12.1.1 Barycentric Interpolation

The idea of $O(N \log N)$ is to exploit the barycentric formula for polynomial interpolation. Let us derive the formula and see how it helps.

First, introduce the quantity $\gamma(x) = \prod_{j=0}^{N-1}(x - x_j)$. Now note that the Lagrange basis polynomials $\ell_j(x)$ can be rewritten as:

$$\ell_j(x) = \gamma(x) \cdot \frac{w_j}{x - x_j}, \quad w_j = \frac{1}{\prod_{k=0, k\neq j}^{N-1}(x_j - x_k)}, \quad j \in [N]$$

Remark. This step might seem unobvious, so let us be careful here. Let us see why expression $\gamma(x) \cdot \frac{w_j}{x - x_j}$ indeed gives the desired basis polynomial:

$$\gamma(x) \cdot \frac{w_j}{x - x_j} = \left(\prod_{k=0}^{N-1}(x - x_k)\right) \cdot \frac{1}{\left(\prod_{k=0, k\neq j}^{N-1}(x_j - x_k)\right)(x - x_j)}$$

Note that we can cancel out $(x - x_j)$ from both numerator and denominator and get

$$\left(\prod_{k=0, k\neq j}^{N-1}(x - x_j)\right) \cdot \frac{1}{\prod_{k=0, k\neq j}^{N-1}(x_j - x_k)} = \prod_{k=0, k\neq j}^{N-1} \frac{x - x_j}{x_j - x_k} = \ell_j(x),$$

which is exactly what we wanted to show.

Good, so why do we even need such an expression for ℓ_j? Let us substitute it back into the interpolation formula:

$$p(x) = \sum_{i=0}^{N-1} a_i \ell_i(x) = \sum_{i=0}^{N-1} a_i \gamma(x) \cdot \frac{w_i}{x - x_i} = \gamma(x) \sum_{i=0}^{N-1} \frac{w_i}{x - x_i} a_i$$

In regards to this formula, we give the following definition.

Proposition 12.1. The **barycentric interpolation formula** for the interpolation problem $p(x_j) = a_j, j \in [N]$, given by $p(x) = \gamma(x) \sum_{i \in [N]} \frac{w_i}{x - x_i} a_i$ with $\gamma(x) = \prod_{i \in [N]} (x - x_i)$, requires $O(N)$ operations to compute and $O(N^2)$ operations to pre-compute.

Proof. Coefficients $\{w_j\}_{j \in [N]}$ are independent of x, and so are the values $\{a_j\}_{j \in [N]}$. To compute $\{w_j\}_{j \in [N]}$, one needs $O(N)$ operations for each w_j, and thus $O(N^2)$ operations in total. To compute the polynomial $p(x)$, one needs $O(N)$ operations to compute $\gamma(x)$ and $O(N)$ operations to compute the sum, knowing $\{w_j a_j\}_{j \in [N]}$. □

Of course, in reality, storing N values $\{w_j\}_{j \in [N]}$ requires $O(N)$ memory, which is not optimal. Moreover, these points on their own are useless and typically are not used in any other parts of the protocol. This is where the ***N*th roots of unity** come into play.

12.1.2 Multiplicative Cyclic Subgroup

Again, assume we have the prime field $\mathbb{F}_p$. Let ω be a **primitive** N**-th root of unity**, i.e., $\omega^N = 1$ and $\omega^j \neq 1$ for $j < N$. The set $\Omega = \{\omega^j\}_{0 \leq j < N}$ is called the N**-th root of unity subgroup** of $\mathbb{F}_p$ of order N. One might ask the following question: why such primitive root even exists? Consider the following lemma, briefly mentioned in Section 3.1.3.

Lemma 12.2. For $\mathbb{F}_p$ there exists a primitive N-th root of unity if and only if $N \mid (p - 1)$.

What is so special about the set Ω? The magic of Ω is that it allows to compute certain polynomial operations (such as interpolation or multiplication) using the **Discrete Fourier Transform** (DFT) or, equivalently, the **Number Theoretic Transform** (NTT) algorithm. Consider the first central lemma.

Lemma 12.3. The vanishing polynomial of the set Ω is given by $z_\Omega(X) = X^N - 1$.

Proof Idea. If ω is the Nth primitive root, then for any $h \in \Omega$ we have $h^N = 1$ and therefore all elements of Ω are the roots of $X^N - 1$. There are precisely N such roots, so $X^N - 1$ can be decomposed as a product of linear factors $c \cdot \prod_{j=0}^{N-1} (X - \omega^j)$. It is easy to see that $c = 1$ by comparing the leading coefficient.

Now, let us come back to the barycentric formula. We have seen that the interpolation polynomial $p(x)$ can be written as $p(x) = \gamma(x) \sum_{i=0}^{N-1} \frac{w_i}{x - x_i} a_i$. The key idea of the FFT is to set $x_j = \omega^j$. What does it give us? We give the following proposition.

Proposition 12.4. Suppose the interpolation domain is chosen so that $a_i = \omega^i$. Then, following the notation of Proposition 12.1, certain expressions simplify to the following:

- $\gamma(x) = x^N - 1$.
- $w_i = \omega^i / N$.

Proof Idea. For the first claim, notice that by definition $\gamma(x) = \prod_{i=0}^{N-1}(x - \omega^i)$, which is exactly the vanishing polynomial of Ω. Thus, $\gamma(x) = z_\Omega(x) = x^N - 1$ from Lemma 12.3.

As for the second claim, recall that $w_i = 1/\prod_{j\neq i}^{N-1}(\omega^i - \omega^j)$. Intuitively, the real analysis shows that $w_i = 1/\gamma'(x_i)$ and since $\gamma(x) = x^N - 1$, we have exactly $w_i = 1/Nx^{N-1}\Big|_{x=\omega^i} = \omega^i/N$. The same result can be obtained by direct computation. □

That being said, the barycentric formula is now given by:

$$p(x) = \frac{x^N - 1}{N} \sum_{j\in[N]} \frac{\omega^j}{x - \omega^j} a_j$$

This formula is a much more convenient form for the computation of the interpolation polynomial! First, the evaluation of the sum requires $O(N)$ operations and it depends only on the values $\{\omega^j\}_{j\in[N]}$, which are typically pre-computed and used in many other parts of the protocol. Second, the evaluation of the vanishing polynomial $x^N - 1$ requires $O(\log N)$ operations using the fast exponentiation algorithm, compared to naive $O(N)$ operations.

12.1.3 Fast Polynomial Multiplication

Forward NTT. Using the Nth roots of unity Ω, we can also compute the polynomial multiplication in $O(N \log N)$ operations. The idea is to use the **Number Theoretic Transform** (NTT) algorithm, which is a generalization of the Fast Fourier Transform (FFT) to the finite fields. But first, let us define what NTT is.

Definition 12.5 (NTT)**.** Suppose the polynomial is given by $p(x) = \sum_{j=0}^{N-1} p_j x^j \in \mathbb{F}[x]$. In its essence, the polynomial is defined as a vector of coefficients $\boldsymbol{p} = (p_0, \ldots, p_{N-1})$. The **Number Theoretic Transform** (NTT) of polynomial $p(x)$ is the vector of evaluations at the Nth roots of unity Ω: $\mathsf{NTT}(\boldsymbol{p}) = (p(\omega^0), p(\omega^1), \ldots, p(\omega^{N-1}))$.

Remark. Typically, the jth component of the NTT vector is denoted as $\hat{p}_j = \mathsf{NTT}(\boldsymbol{p})_j$.

If computed naively, the NTT requires $O(N^2)$ operations, since each evaluation of $p(x)$ requires $O(N)$ operations. However, due to the specifics of the selected domain Ω, the NTT can be computed in $O(N \log N)$ operations. Let us emphasize this in the following lemma.

Lemma 12.6. The **Number Theoretic Transform** (NTT) of a polynomial $p(x)$ can be computed in $O(N \log N)$ operations using the Nth roots of unity. This is possible only if the prime field $\mathbb{F}_p$ allows to find the primitive 2^k root of unity for $k \in [m]$ with large enough m. Equivalently, $2^k \mid (p-1)$ for $k \in [m]$.

To show this lemma is true, let us develop the concrete algorithm. Notice that our task

consists in computing:

$$p(\omega^i) = \sum_{j\in[N]} p_j(\omega^i)^j = \sum_{j\in[N]} p_j\omega^{ij} \text{ for each } i \in [N]$$

Suppose the considered polynomial is such that $N = 2^r$ (we can always pad the polynomial if that is not the case). Now, let us proceed with the polynomial as follows:

$$p(\omega^i) = \sum_{j=0}^{2^r-1} p_j\omega^{ij} = \sum_{j=0}^{2^{r-1}-1} p_{2j}\omega^{2ij} + \sum_{j=0}^{2^{r-1}-1} p_{2j+1}\omega^{i(2j+1)} = \sum_{j=0}^{2^{r-1}-1} p_{2j}(\omega^{2i})^j + \omega^i \sum_{j=0}^{2^{r-1}-1} p_{2j+1}(\omega^{2i})^j$$

This already looks interesting enough. Notice that we can introduce two new polynomials: $p_E(x) = \sum_{j=0}^{2^{r-1}-1} p_{2j}x^j$ and $p_O(x) = \sum_{j=0}^{2^{r-1}-1} p_{2j+1}x^j$, which are polynomials, containing even and odd coefficients of p, respectively. In that case,

$$p(\omega^i) = p_E(\omega^{2i}) + \omega^i p_O(\omega^{2i})$$

This is quite an interesting observation which already screams divide-and-conquer! However, currently it might still be unclear how to use it: we still have to evaluate N expressions of form $p_E(\omega^{2i}) + \omega^i p_O(\omega^{2i})$ where both polynomials p_E and p_O contain roughly $N/2$ coefficients, totalling in $O(N^2)$ operations again. To counter this, we claim the following: we need only half the domain of Ω to compute both p_E and p_O. To see why, consider the expression $p(\omega^{i+N/2})$:

$$p(\omega^i) = p_E(\omega^{2(i+N/2)}) + \omega^{i+N/2}p_O(\omega^{2(i+N/2)}) = p_E(\omega^{2i}) + \omega^i\omega^{N/2}p_O(\omega^{2i})$$

In other words, having computed $p_E(\omega^{2i})$ and $p_O(\omega^{2i})$, we know not only $p(\omega^i)$, but also $p(\omega^{i+N/2})$ for free! This way, to compute the NTT for N-degree polynomial, we need to evaluate two $\frac{N}{2}$-degree polynomials at $\frac{N}{2}$ points. This way, on each step: (a) the evaluation domain shrinks in half, (b) the complexity of computing polynomials shrinks in half, (c) we get two new polynomials. This way, on each step, we reduce the problem complexity in half!

The reason why the prime field should support multiplicative cyclic subgroups of order 2^k for sufficiently many k is that not always if ω is the Nth primitive root, then ω^2 is the $\frac{N}{2}$th primitive root. If that is not the case, all the aforementioned magic breaks.

We summarize everything so far in the Algorithm 4.

Algorithm 4: Number Theoretic Transform (NTT)

Input : Polynomial $p(x) = \sum_{j=0}^{N-1} p_j x^j$
Output: Vector of evaluations $\mathsf{NTT}(\boldsymbol{p}, \omega)$ at $\Omega = \{\omega\}_{j\in[N]}$

1 **if** $N = 1$ **then**
  Return : (p_0)
2 **end**
3 $H \leftarrow N/2$ /* Compute the domain half-size */
4 $\boldsymbol{p}_E \leftarrow (p_0, p_2, \ldots, p_{N-2})$ /* Find even-indexed coefficients */
5 $\boldsymbol{p}_O \leftarrow (p_1, p_3, \ldots, p_{N-1})$ /* Find odd-indexed coefficients */
6 $\boldsymbol{y}_E \leftarrow \mathsf{NTT}(\boldsymbol{p}_E, \omega^2)$ /* Compute NTT for even polynomial via $\frac{N}{2}$th primitive root ω^2 */
7 $\boldsymbol{y}_O \leftarrow \mathsf{NTT}(\boldsymbol{p}_O, \omega^2)$ /* Compute NTT for odd polynomial via $\frac{N}{2}$th primitive root ω^2 */
Return : $(y_0, \ldots, y_{N-1})$ with $y_j = y_{E,\, j \bmod H} + \omega^j y_{O,\, j \bmod H}$

NTT Domain. OK, so what next? Suppose we want to multiply two polynomials $p(x), q(x) \in \mathbb{F}[X]$ of degree $N = 2^r$ and we have successfully evaluated their NTTs. Say, we got $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$. What can we do next? Here is another trick.

Proposition 12.7. Suppose $m(x) = p(x)q(x)$ is the product of p and q. Then,

$$\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$$

Speaking more formally, $\mathsf{NTT} : (\mathbb{F}^{(\leq N)}[X], \times) \to (\mathbb{F}^N, \odot)$ is a homomorphism between a set of polynomials of degree up to N and their NTT domain. With certain appropriate technicalities, NTT can be extended to the isomorphism.

Intuition. Although this fact might come out of random, we give an intuitive explanation why this holds. One of NTT interpretations is well-known to you *interpolation*. Indeed, polynomials p and q satisfy the following interpolation problem:

$$p(\omega^j) = \mathsf{NTT}(p)_j = \hat{p}_j, \quad q(\omega^j) = \mathsf{NTT}(q)_j = \hat{q}_j.$$

In turn, $\hat{m}_j$ is nothing but the evaluation of m at ω^j. But, if m is the product of p and q and values of p and q at ω^j are $\hat{p}_j$ and $\hat{q}_j$, this immediately implies that $m(\omega^j)$ is nothing but $\hat{p}_j\hat{q}_j$. Meaning, $\hat{m}_j = \hat{p}_j\hat{q}_j$. This, in turn, implies $\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$.

Wow! What this essentially means is that multiplication in NTT domain is very cheap: it requires only $O(N)$ field multiplication operations! Now, the algorithm of multiplication of p and q becomes a little bit more clear at this point:

1. Compute NTTs $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$ of p and q.
2. Compute NTT $\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$ of their product $m = pq$.
3. Restore $m(x)$ from NTT $\hat{\boldsymbol{m}}$ — this problem is called Inverse NTT (INTT).

Inverse NTT. So, the only problem left is restoring the polynomial from its NTT form. This problem is equivalent to solving the interpolation problem:

$$m(\omega^j) = \hat{m}_j, \quad j \in [N]$$

Suprisingly, the **Inverse NTT** is given by $p_j = \frac{1}{N}\sum_{j=0}^{N-1} \hat{p}_j\omega^{-ij}$, which can be computed by running the forward NTT, but with generator ω^{-1}. Division by N is done over $\mathbb{F}_p$, which is surely trivial. We summarize the whole discussion in Figure 12.1.

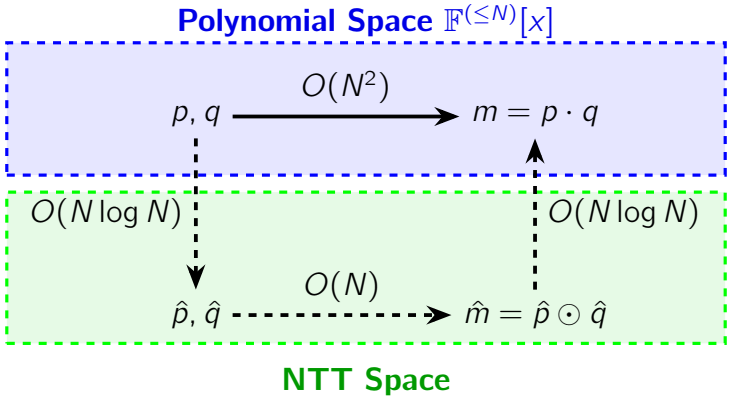

Figure 12.1: Illustration of the NTT Algorithm