Plookup in action

Ariel Gabizon   Zachary J. Williamson

Turbo-PLONK programs (based on PLONK[GWC])

a_1	b_1	c_1	d_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_i	b_i	c_i	d_i
a_{i+1}	b_{i+1}	c_{i+1}	d_{i+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$

- Local low-degree constraints between rows (e.g. $\mathbf{a}_{i+1} = \mathbf{b}_i^2 + \mathbf{c}_i$)
- Global equality constraints between any two cells (e.g. $\mathbf{a}_{100} = \mathbf{d}_2$).

Ultra-PLONK programs

- Local low-degree constraints between rows (e.g. $a_{i+1} = b_i^2 + c_i$).
- Global equality constraints between any two cells (e.g. $a_{100} = d_2$).
- **Lookup constraints** - e.g. (a_5, b_5, c_5) is contained in the rows of a predefined table T.

Lookup constraints in SNARKs

First used in Arya[Bootle, Cerulli, Groth, Jakobsen, Maller]

Lookup constraints in SNARKs

First used in Arya[Bootle, Cerulli, Groth, Jakobsen, Maller]

Plookup [GW20] gives improved efficiency:
$2(|\mathbf{T}| + |\boldsymbol{w}|)$ prover group exp

$|\mathbf{T}|$ - number of rows in table
$|\boldsymbol{w}|$ - length of witness

Example: bitwise XOR with "direct" table

For row values $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ want to show $\mathbf{c} = \mathbf{a} \oplus \mathbf{b}$ as 11-bit strings.

Example: bitwise XOR with “direct” table

For row values $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ want to show $\mathbf{c} = \mathbf{a} \oplus \mathbf{b}$ as 11-bit strings.

Use table $\mathbf{T}$ of all triplets $(\mathbf{a}, \mathbf{b}, \mathbf{c})$
s.t. $\mathbf{c} = \mathbf{a} \quad \oplus \quad \mathbf{b}$.

Example: bitwise XOR with "direct" table

For row values $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ want to show $\mathbf{c} = \mathbf{a} \oplus \mathbf{b}$ as 11-bit strings.

Use table $\mathbf{T}$ of all triplets $(\mathbf{a}, \mathbf{b}, \mathbf{c})$
s.t. $\mathbf{c} = \mathbf{a} \quad \oplus \quad \mathbf{b}$.

$|\mathbf{T}| = 2^{22}$

Another approach - Sparse representations

Table T_1 of pairs (α, α_s) - α is 10-bit string, α_s is "α with zeroes in between bits" -

$$\alpha = \Sigma \alpha_i \cdot 2^i, \alpha_s = \Sigma \alpha_i \cdot 4^i \tag{1}$$

Another approach - Sparse representations

Table T_1 of pairs (a, a_s) - a is 10-bit string, a_s is "a with zeroes in between bits" -

$$a = \Sigma a_i \cdot 2^i, a_s = \Sigma a_i \cdot 4^i \tag{1}$$

Field addition on sparse form now gives bitwise XOR :
$a = (11) \qquad b = (10)$

Another approach - Sparse representations

Table T_1 of pairs $(\mathbf{a}, \mathbf{a_s})$ - $\mathbf{a}$ is 10-bit string, $\mathbf{a_s}$ is "$\mathbf{a}$ with zeroes in between bits" -

$$\mathbf{a} = \boldsymbol{\Sigma}\mathbf{a_i} \cdot 2^i, \mathbf{a_s} = \boldsymbol{\Sigma}\mathbf{a_i} \cdot 4^i \qquad (1)$$

Field addition on sparse form now gives bitwise XOR :

$\mathbf{a} = (11) \qquad \mathbf{b} = (10)$

$\mathbf{a_s} = (0101)$

$\mathbf{b_s} = (0100)$

Another approach - Sparse representations

Table T_1 of pairs (a, a_s) - a is 10-bit string, a_s is "a with zeroes in between bits" -

$$a = \Sigma a_i \cdot 2^i, a_s = \Sigma a_i \cdot 4^i \tag{1}$$

Field addition on sparse form now gives bitwise XOR :

$\mathbf{a} = (11) \qquad \mathbf{b} = (10)$

$\mathbf{a}_s = (0101)$

$\mathbf{b}_s = (0100)$

$\mathbf{a}_s + \mathbf{b}_s = (1001)$

Another approach - Sparse representations

Table T_1 of pairs $(\mathbf{a}, \mathbf{a}_s)$ - $\mathbf{a}$ is 10-bit string, $\mathbf{a}_s$ is "$\mathbf{a}$ with zeroes in between bits" -

$$\mathbf{a} = \Sigma \mathbf{a}_i \cdot 2^i, \mathbf{a}_s = \Sigma \mathbf{a}_i \cdot 4^i \tag{1}$$

Field addition on sparse form now gives bitwise XOR :

$\mathbf{a} = (11) \qquad \mathbf{b} = (10)$

$\mathbf{a}_s = (0101)$

$\mathbf{b}_s = (0100)$

$\mathbf{a}_s + \mathbf{b}_s = (1001)$

Odd bits are XORs

Another approach - Sparse representations

After adding in sparse form, can use another lookup to "decode" XOR result $T_2 = \{c_s, c_{XOR}\}$ so

$$c_s = \Sigma c_i 4^i, c_{XOR} = \Sigma \phi(c_i) 4^i,$$

$$\phi(0) = 0, \phi(1) = 1, \phi(2) = 0, \phi(3) = 1$$

Another approach - Sparse representations

After adding in sparse form, can use another lookup to "decode" XOR result $T_2 = \{\mathbf{c}_s, \mathbf{c}_{XOR}\}$ so

$$\mathbf{c}_s = \Sigma \mathbf{c}_i 4^i, \mathbf{c}_{XOR} = \Sigma \phi(\mathbf{c}_i) 4^i,$$

$$\phi(0) = 0, \phi(1) = 1, \phi(2) = 0, \phi(3) = 1$$

Can get AND at same time (see Arya paper)

SHA-256 with Sparse representations on Steroids

$\mathbf{MAJ}'$ is one of the two main “chunks” of a SHA round:

- $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ 32-bit values
- $\ggg$ is right rotation.

$\mathbf{MAJ'}$ is one of the two main "chunks" of a SHA round:

- $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ 32-bit values
- $\ggg$ is right rotation.

$$\mathbf{MAJ'}(\mathbf{a},\mathbf{b},\mathbf{c}) :=$$

$$(\mathbf{a} \ggg 2) \oplus (\mathbf{a} \ggg 13) \oplus (\mathbf{a} \ggg 22) \oplus \mathbf{MAJ}(\mathbf{a},\mathbf{b},\mathbf{c})$$

$\mathbf{MAJ}'$ is one of the two main "chunks" of a SHA round:

- $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 32-bit values
- $\ggg$ is right rotation.

$$\mathbf{MAJ}'(\mathbf{a}, \mathbf{b}, \mathbf{c}) :=$$

$$(\mathbf{a} \ggg 2) \oplus (\mathbf{a} \ggg 13) \oplus (\mathbf{a} \ggg 22) \oplus \mathbf{MAJ}(\mathbf{a}, \mathbf{b}, \mathbf{c})$$

We map $\mathbf{a}, \mathbf{b}, \mathbf{c}$ into 16-sparse form:
$\Sigma a_i 2^i \rightarrow \Sigma a_i 16^i$

We map $\mathbf{a}, \mathbf{b}, \mathbf{c}$ into 16-sparse form:
$\Sigma \mathbf{a}_i 2^i \rightarrow \Sigma \mathbf{a}_i 16^i$

In sparse form we simply add in field:

$$4*((\mathbf{a} \ggg 2)+(\mathbf{a} \ggg 13)+(\mathbf{a} \ggg 22))+(\mathbf{a}+\mathbf{b}+\mathbf{c})$$

We map $\mathbf{a}, \mathbf{b}, \mathbf{c}$ into 16-sparse form:
$\Sigma \mathbf{a}_i 2^i \to \Sigma \mathbf{a}_i 16^i$

In sparse form we simply add in field:

$$4 * ((\mathbf{a} \ggg 2) + (\mathbf{a} \ggg 13) + (\mathbf{a} \ggg 22)) + (\mathbf{a} + \mathbf{b} + \mathbf{c})$$

Addition result is "injective enough" to retrieve output of $\mathbf{MAJ'}$.

Getting the rotations

Split 32-bit $\mathbf{a}$ to limbs $(\mathbf{a}_2, \mathbf{a}_1, \mathbf{a}_0)$ of $10, 11, 11$ bits respectively.

Getting the rotations

Split 32-bit a to limbs (a_2, a_1, a_0) of $10, 11, 11$ bits respectively.

We have in total 9 "rotate contributions": 3 right-rotates - $2, 13, 22$ of the three limbs.

Getting the rotations

Split 32-bit $\mathfrak{a}$ to limbs $(\mathfrak{a}_2, \mathfrak{a}_1, \mathfrak{a}_0)$ of $10, 11, 11$ bits respectively.

We have in total 9 "rotate contributions": 3 right-rotates - $2, 13, 22$ of the three limbs.

But only two "non-trivial" contributions: $(\mathfrak{a}_1, 13), (\mathfrak{a}_0, 2)$

Getting the rotations

Split 32-bit $\mathfrak{a}$ to limbs $(\mathfrak{a}_2, \mathfrak{a}_1, \mathfrak{a}_0)$ of $10, 11, 11$ bits respectively.

We have in total 9 "rotate contributions": 3 right-rotates - $2, 13, 22$ of the three limbs.

But only two "non-trivial" contributions:
$(\mathfrak{a}_1, 13), (\mathfrak{a}_0, 2)$
both can be computed with a table of right rotate by 2.

Getting the rotations

Split 32-bit $\mathfrak{a}$ to limbs $(\mathfrak{a}_2, \mathfrak{a}_1, \mathfrak{a}_0)$ of $10, 11, 11$ bits respectively.

We have in total 9 "rotate contributions": 3 right-rotates - $2, 13, 22$ of the three limbs.

But only two "non-trivial" contributions:
$(\mathfrak{a}_1, 13), (\mathfrak{a}_0, 2)$
both can be computed with a table of right rotate by 2.

In total for $\mathbf{MAJ'}$- 3 tables of size $\leq 2^{11}$