Application of Graph Methods for Efficient Quotient Polynomial Evaluation in Halo2

Karthik Inbasekar
Ingonyama
karthik@ingonyama.com

Roman Palkin
Ingonyama
roman@ingonyama.com

Guy Weissenberg
EPFL
guy.weissenberg@epfl.ch

Abstract. In this work we present our research on applications of graph methods to parallelize the quotient polynomial in the Halo2 Zero Knowledge Proof (ZKP) system. The evaluation of the quotient polynomial has a complex data-interdependency which makes it a serious compute and memory bottleneck in halo2 applications. We achieve partitioning of quotient polynomial evaluation for large circuits, into groups of connected components such that the groups are weakly connected. The weak connections are data that can be copied to the groups at the time of evaluations thus parallelizing the problem and solving data-inter dependency. As a concrete Proof of concept, we demonstrate an efficient partition of quotient polynomial evaluation into two independent bins for the Taiko ZKEVM super circuit in CPU. In a two GPU setup, the two bins can be run completely independent in parallel, thus reducing latency and memory significantly.

1 Introduction

Halo2 is a comprehensive Zero Knowledge Proof system used in several production level environments today [1, 2]. Halo2 provides a complete framework to create and prove circuits based on plonkish arithmetization with circuit constraints expressed using custom gates, and lookup tables. This allows significant flexibility in the stages of ZK design, from arithmetization, optimizing constraints, proving methods and commitment schemes. The original Halo2 was built by the Zcash team [3, 4] and featured a Inner Product Argument scheme for polynomial commitment and opening [5, 6].

In this article, we provide some insights on computational aspects in the halo2 proving system augmented by the KZG [7] commitment scheme popularly known as the PSE-KZG fork [1]. The main objective of this article is to explore efficient ways to evaluate the most non-universal part in the proof generation process, namely the quotient polynomial. The proof system in halo2 verifies the satisfiability of three mathematically independent conditions. These are the gate constraints, copy/permutation constraints and lookup constraints. While the constraints are independent, the data (columns) that participates in these constraints overlap and create a complex data dependency as shown in fig. 1. A given column can participate in a degree 10 gate constraint, in a degree 2 lookup constraint and be part of a permutation set. This makes the constraint evaluation part complex, inefficient and makes it difficult to create circuit agnostic optimizations at the hardware level. Moreover, much of the data dependency complexity is built at the level of circuit construction itself.

[1] https://github.com/privacy-scaling-explorations/halo2

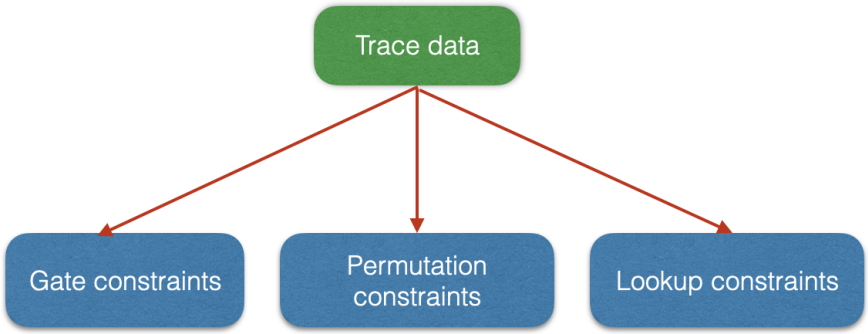

Figure 1. Data dependency in the quotient polynomial

When it comes to proving time, parallelizability is crucial for any hardware efficient solution. A highly parallelized design can lead to significant performance improvements when one can also balance some of the additional space complexity that is created due to data flow and memory. A simple strategy to solve space complexity is to use multiple hardware acceleration devices. Apart from being subject to cost, major problems that appear in this approach are data inter-dependency and device occupancy.

1.1 Technical overview

The first problem we address is efficient evaluation of the quotient polynomial. For this we turn to Connected Components Analysis (CCA) from graph theory. A connected component in an un-directed graph refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group. We obtain the symbolic representation of the circuit constraint system and build a data dependency graph for any given circuit. We use a connected component evaluation on the resultant graph to estimate closed sub expressions. We present the algorithm and the obtained results in §4.1. Best case scenarios are when we have several disjoint connected components and each disjoint component can be evaluated in parallel, though this rarely happens in many practical circuits such as ZKEVMs. Although the result is very circuit dependent, in general the permutation constraints (and to some extent lookups) (see §A) interlink several columns and quite often we end up with a single connected component for large circuits with many permutations. Nevertheless the connected component approach could be used as an intermediate step in a computation which is separable using other methods such as community detection.

Community detection addresses the problem of global seperability of a graph. For problem seperability in quotient polynomial evaluation, we apply the famous Girvan-Newman (GN) algorithm [8]. Intuitively, GN uses the simple observation that edges that connect communities in a graph have maximum betweenness of all edges in a graph (see §4.3).[2] In this definition, communities in a graph are groups of connected components such that the

[2]We provide a rust implementation of the algorithm (3) in https://github.com/ChickenLover/petgraph/blob/master/src/algo/community.rs

groups themselves are weakly connected. We apply the algorithm on the data dependency graph and try to separate communities with minimal overlapping columns. As a concrete example, we illustrate a Proof Of Concept (CPU) of problem separability using GN algorithm for the Taiko ZKEVM circuit. We report an effective parallelization strategy into two communities. The results are summarized in §4.4. Thus, in a 2-GPU system, both the communities can be run in parallel. The implemented POC passes the correctness test, (same quotient polynomial as the native algorithm). Encouraged by these results we tried the same strategy and applied it to the Scroll ZKEVM circuit (§4.5). We have found that in Scroll case, our approach leads to inefficient partitions and we discuss why it is difficult to do so. Our observation is that if the largest community in a graph is bigger than 60% of the size of the circuit, it is much harder to find efficient partitions. In the case of Taiko, the largest community is 54% of the total circuit size, while for Scroll, the largest community is 68% of the total circuit size. We believe that in the case of Scroll, the clustering in this way is due to large number of permutations (apart from gates usage, and lookups), for example Scroll is over 190 columns out of 971 for permutations as opposed to Taiko which has 15 columns out of 552 for permutations.

Finally, no community detection is satisfying without pretty pictures, and there are several useful community detection and visualization algorithms in Graphviz [9] which are based on different graph visualization algorithms. We apply gvmap [3] and sfdp [4] accessible via graphviz visualization software [9] on the Taiko and Scroll super-circuit graphs and the results are available in §4.6. We find that the results are consistent with our analysis based on GN in §4.5 and §4.4. A key takeaway is that these heuristic community methods indicate crucial efficiency factors in circuit design that can inhibit factorizability and affect proving complexity for the quotient polynomial.

2 Background: A Bird's Eye View of Halo2

In fig. 2 we present an overview of halo2 from a high level, we briefly touch up on some definitions and highlight issues from a computational perspective to make the article self contained. [5]

2.1 Arithmetization

Arithmetization is the art of converting a circuit into a matrix representation of table entries that accurately (and hopefully efficiently) defines the state and state transitions of the circuit. State refers to the relationship between private, public inputs, intermediate values of the circuit at a defined instant. Halo2 uses the plonkish arithmetization defined by a matrix $M_{n\times m}$ consisting of cells, with each cell being an element in a finite field $\mathbb{F}$.

[3] gvmap is a graph visualization algorithm that finds clusters and creates a geographical map highlighting the clusters

[4] sfdp stands for Scalable forced directed placement, which is useful for fast visual rendering of large graphs.

[5] More comprehensive reviews and resources on halo2 can be found in https://ingonyama-zk.github.io/ingopedia/.

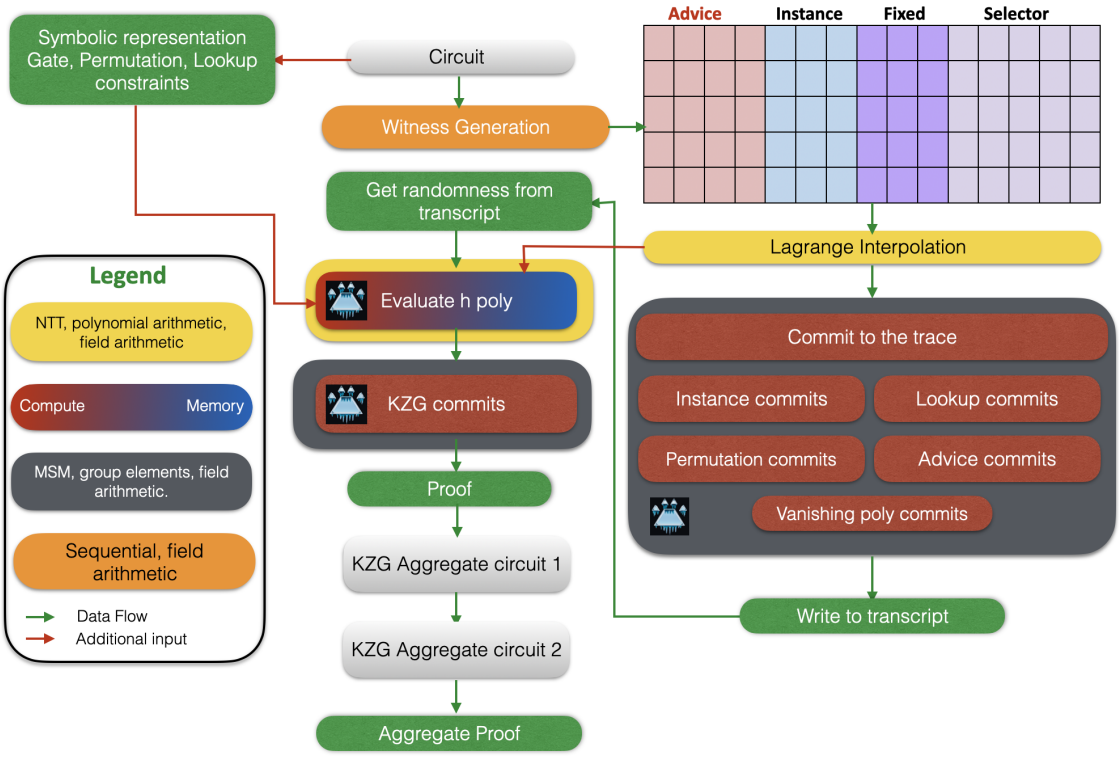

Figure 2. Halo2 a Bird's eye overview: In general, the red blobs are parallelizable compute bottlenecks that can be solved potentially with GPU solutions. The exception is witness generation. The blue coloration indicates memory bottlenecks, which can arise due to large degree constraints or inefficient computation in h-poly.

Here $n = 2^k$ (for $k \in \mathbb{Z}$) is the order of a multiplicative subgroup in a finite field $\mathbb{F}$. Columns in the table in fig. 2 are classified as

- **Advice:** $a_i \in \mathbb{F}^n$: Private witness values known only to the prover.
- **Instance:** $I \in \mathbb{F}^n$ Public input common to both prover and verifier.
- **Fixed:** $f_i \in \mathbb{F}^n$ Public constants used in circuit common to both prover and verifier.
- **Selector:** $q_i \in \{0, 1\}$: Public Circuit configuration, common to prover and verifier.

The intermediate values are recorded in the advice columns a_i and the process of generating these values is called witness generation. Witness generation is a mostly serial process and in general involves arithmetic in scalar field as well as base field. We use the notation $v_i \in \{a_i, I, f_i, q_i\}$ to collectively denote all the columns of the matrix M. The elements in the trace table in fig. 2 are constrained by:

- **Gate constraints:** These are constraints over the advice and fixed columns which are enabled/disabled by a boolean value in the selector columns.

$$q_j \cdot C_j(\{a_i\}, \{f_i\}) = 0 \tag{2.1}$$

In general the constraints can apply simultaneously to groups of contiguous rows in the matrix.[6] A simple example of incomplete elliptic curve addition of two affine points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ giving the result $R = (x_3, y_3)$ is expressed as two constraints with a boolean selector q_{EC}

$$\begin{aligned} q_{EC} \cdot ((x_3 + x_2 + x_1) \cdot (x_2 - x_1)^2 - (y_2 - y_1)^2) = 0 \\ q_{EC} \cdot ((y_3 + y_2) \cdot (x_1 - x_2) - (y_1 - y_2) \cdot (x_2 - x_3)) = 0 \end{aligned} \tag{2.2}$$

- **Permutation/equality constraints:** This enforces wiring of the circuit

$$P_j(\{a_i\}, \{\sigma(a_i)\}) = 1 \tag{2.3}$$

by applying the condition that specific groups of cells that are connected by wires have the same value. For instance, if the cell with index (a, b) is wired to (copied to cell index (c, d) and (e, f) then the set $S : \{M_{a,b}, M_{c,d}, M_{e,f}\}$ are invariant under all permutations $\sigma(S)$.

- **Lookup constraints:** If a lookup table is used in the verification of a constraint the corresponding cell values in the advice columns are constrained to be in a set membership

$$q_j \cdot L_j(\{a_i\}, \{f_i\}) = 0 \tag{2.4}$$

in some column of the lookup table. Like a custom gate, the lookup argument is enabled with a selector. Each column in the lookup table is a designated fixed column f_i in table in fig. 2.

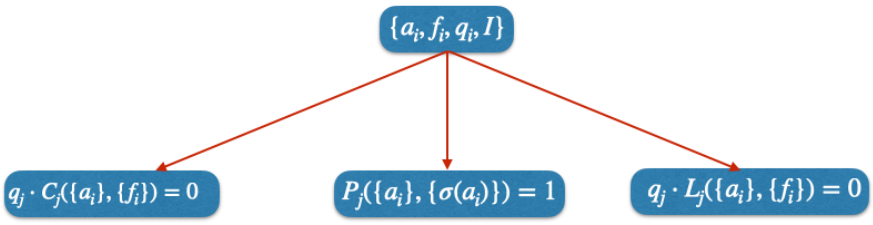

Figure 3. Data flow dependency generated at the arithmetization level

We emphasize that the gate (2.1), copy (2.3) and lookup (2.4) constraints are mathematically independent. However, from a computational perspective, a given column, in general, participates in each of these mathematically independent constraints and in different degrees after interpolation. For instance, a given advice column can participate in a high degree gate constraint, an independent lookup constraint and also be part of a permutation column set. This creates a problem of strong data dependency as shown in fig. 3, which determines the overall computational complexity of the constraints evaluation stage which is the main subject of §4.

[6] There are several optimization features such as selector combination that affect the table sizes and degree complexity, and we do not cover them in this discussion. We refer to the halo2book [3] for more details.

2.2 Lagrange interpolation

Once the advice columns are filled in the witness generation phase, a polynomial representation of the columns is constructed using Lagrange interpolation or Number Theoretic Transforms (NTT) [10]. In the table in fig. 4, let the columns in the trace-table be denoted as $(v_0, v_1, \dots v_{m-1}) \in \mathbb{F}^{n \times m}$, where $|H| = n$ is the domain size of the vector v_i. Let the domain be indexed using the roots of unity in the domain $H = \{1, \omega, \dots \omega^{n-1}\}$ as shown in fig. 4 such that

v_0	v_1				v_{m-1}
$v_0(\omega^0)$	$v_1(\omega^0)$	$\cdots$	$\cdots$	$\cdots$	$v_{m-1}(\omega^0)$
$v_0(\omega^1)$	$v_1(\omega^1)$	$\cdots$	$\cdots$	$\cdots$	$v_{m-1}(\omega^1)$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\vdots$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\vdots$
$v_0(\omega^{n-1})$	$v_1(\omega^{n-1})$	$\cdots$	$\cdots$	$\cdots$	$v_{m-1}(\omega^{n-1})$

Figure 4. Lagrange interpolation

$$v_i(\omega^j) = M_{j,i} \ \forall j \in \{0, 1, \dots n-1\} \tag{2.5}$$

where ω is a generator is the nth root of unity. The polynomial representation of these vectors are written using the Lagrange basis defined by

$$\chi_{x_i}(X) = \frac{\prod_{k=0, k\neq i}^{n-1}(X - x_k)}{\prod_{k=0, k\neq i}^{n-1}(x_i - x_k)} \tag{2.6}$$

that satisfy the relations

$$\chi_{x_i}(x_j) = \delta_{ji} = \begin{cases} 1 \ , \ j = i \ \forall \ i, j = 0, 1, \dots n-1 \\ 0 \ , \ \text{otherwise} \end{cases} \tag{2.7}$$

for all $x_i \in H$.

$$v_i(X) = \sum_{j=0}^{n-1} M_{ji} \cdot \chi_{x_j}(X) \tag{2.8}$$

Using the polynomial representation, all the circuit constraints (2.1), (2.3) and (2.4) are expressed as polynomial identities point-wise satisfied in the domain H. The interpolated polynomials are used later in the constraints evaluation stage and need to be stored. Part of this computation includes interpolation of pre-computed circuit-dependent data of the fixed f_i and selector q_i columns, which are part of the setup.

2.3 Commit to trace

Following the interpolations, the trace values are committed using MSMs (Multi Scalar Multiplication. MSM is a highly parallelizable problem with several solution variants based on the Pippenger algorithm[7]. In the particular case of Halo2, the Structured Reference String (SRS) is common to all the columns and the size of the SRS degree bounds the number of rows n in the table in fig. 2. The only data that is required from this point of the computation at the end of the commit-to-trace is a transcript of group elements and random challenges for computing the quotient polynomial and the vanishing argument.

2.4 Quotient poly and vanishing argument

Let $\phi_i(v_0, v_1, \ldots, v_{m-1})$ denote a constraint. Each constraint ϕ_i is a polynomial evaluation of an independent, gate,a lookup or a permutation constraint. The composition polynomial is a randomized linear combination of such monomials expressed as

$$\Phi(X) = \sum_c r^c \cdot \phi_c(X) \ \forall X \in H \tag{2.9}$$

Since for $\Phi(X)$ to make sense as a polynomial of degree $d > n$ it must have $d+1$ evaluations. The computation part of this step, is dominated by NTTs and iNTTs and is non universal in complexity. The complexity can become compute/memory or both and with large circuits, memory bottlenecks are frequent. Nevertheless due to several advances in NTT computations in GPU the compute bottleneck is efficiently solvable. Quite so often the memory bottleneck is much harder to solve, and it further results in reduced performance due to heavy reads/writes between the host and the device.

The main factors that affect the efficiency of evaluating $\Phi(X)$ are:

- Degree distribution of constraints. The highest degree constraint determines the degree of $\Phi(X)$ and hence presence of many large degree constraints is equal to many large degree convolutions or several convolutions with several small CosetNTT's.
- Presence of large number of common sub expressions, can make use of evaluation strategies to store and reuse subexpressions across monomials as much as possible.
- Parallelizability: In some cases, the circuit is simply too large (wide), e.g. a ZKEVM circuit. This makes it nearly impossible to run it on a single device. In such cases, a useful strategy is to find disjoint partitions with no overlap. In reality this does not happen, and the best case scenario is when there is minimal overlap of partitions. This effectively parallelizes quotient polynomial into smaller communities with minimal overlapping columns.

The vanishing argument essentially ensures that the constraint relations (2.9) are pointwise satisfied in the domain H.

$$h(X) = \frac{\Phi(X)}{X^n - 1} \tag{2.10}$$

[7]See Ingopedia: https://ingonyama-zk.github.io/ingopedia/ in MSM section for more references

Constraint degree	Number of columns	NTT size
n	122	2^{19}
$3n$	104	2^{21}
$4n$	43	2^{21}
$6n$	29	2^{22}
$9n$	36	2^{23}
$90n$	380	2^{26}

Table 1. Skewed constraint degree distribution

Note that in general the degree of $h(X)$ is much larger than the SRS length n and in general there is a large INTT at the end of the computation in order to reduce

$$h(X) = \tilde{h}_0(X) + X^n \tilde{h}_1(X) + \dots \tag{2.11}$$

it into chunks the size of the SRS.

2.5 KZG commitments

The main computations in this part are commitments of the several pieces (2.11) of the quotient polynomial from h-poly. The KZG quotients (see for example [11]) are computed and committed using MSMs, and one can use a batch opening algorithm for several of the committed polynomials at the random challenge point. This is again something that can be very easily handled with a GPU and is not so much a bottleneck.

2.6 KZG aggregation

The KZG aggregation scheme allows to aggregate several KZG proofs due to their additive homomorphism and a single verification is sufficient to verify a batch of aggregated proofs. The aggregate circuit, essentially shows that the accumulation (EC addition) of each of the left, right limbs of the pairing equation is done correctly. The degree of the constraints are at most degree 4 (for incomplete EC addition) (2.2). In this case, the computational complexity is dominated by the commitments, and quotient polynomial is a secondary bottleneck. This is again a GPU problem with an efficient universal solution. See for instance our ICICLE GPU solution [8] for the ezkl aggregate circuit [9].

3 Quotient Polynomial and Strategies For Evaluation

In this section, we dive into the details of the quotient polynomial computation.

3.1 Smash and crash

The data flow diagram is described below in fig. 5. This strategy is only suitable for small circuits with the degrees of constraints somewhat homogeneous.

[8] https://github.com/ingonyama-zk/icicle
[9] https://blog.ezkl.xyz/post/acceleration/

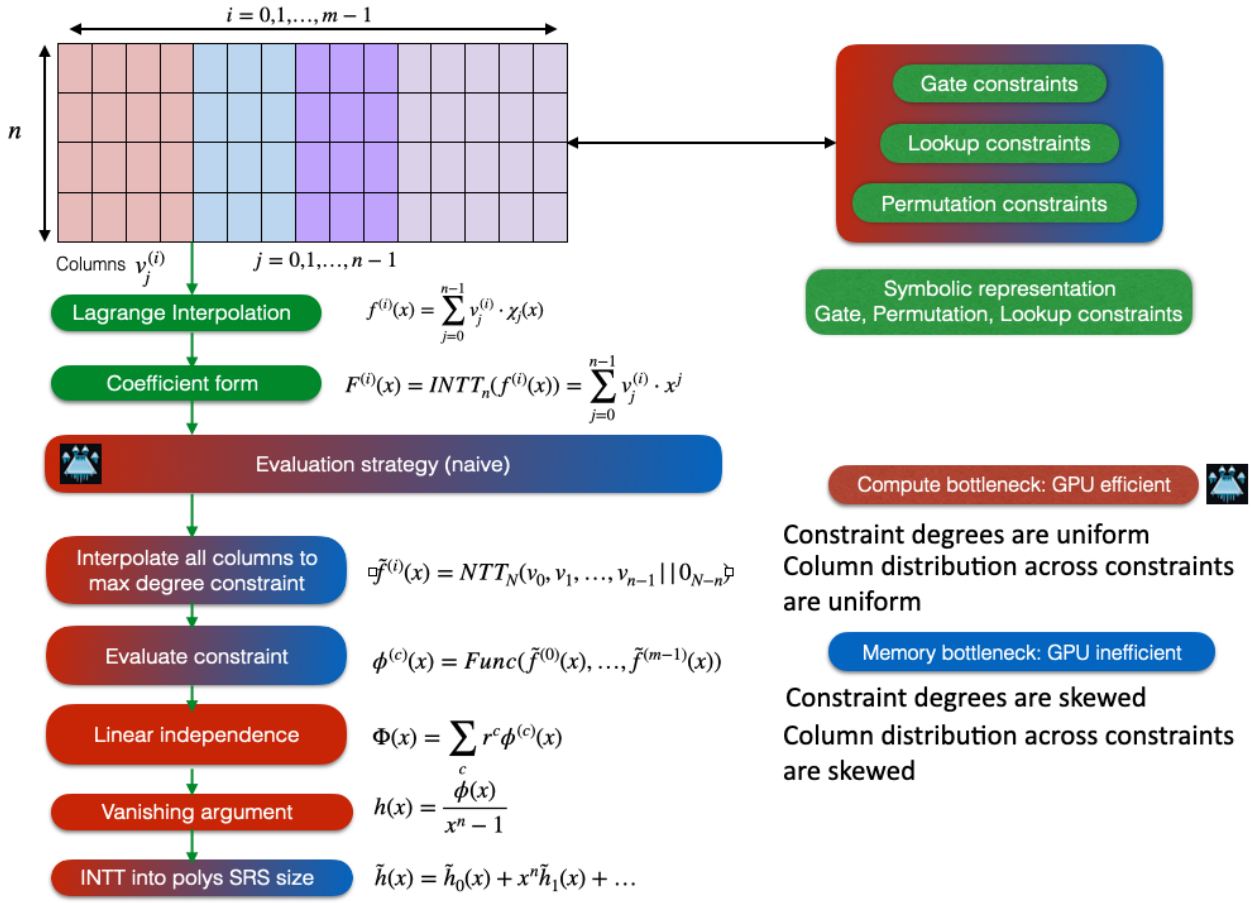

Figure 5. Quotient polynomial, data flow diagram. This diagram simply represents the most basic strategy for evaluation of the h-poly constraints

For a typical circuit with configuration as in table 1, the strategy presented in fig. 5 would be a terrible idea. To interpolate all columns to $90n$ would result in a severe memory bottleneck in a GPU computation. Rather, it is efficient to evaluate all smaller constraints and do fewer large size NTTs as much as possible. Although one could argue that with cosetNTT, larger sized NTTs translate to a large number of small sized cosetNTTs which are more parallelizable, the memory bottleneck still remains.

An optimization suggested by ZCash [10] is to use the Abstract Syntax Tree (AST) and represent the computation as Directed Acyclic Graphs (DAG) for optimal constraint evaluation. This is much harder to implement for a general case. The strategy described in fig. 5 is still good enough for small circuits or when majority of the constraints are of a similar degree, such as a circuit that uses lookup arguments, on the same column, and can be optimized very well following the AST-DAG approach.

[10] https://github.com/zcash/halo2/issues/427

3.2 Divide and conquer

This proposal was originally due to zcash[11], was implemented by Scroll[12] and further utilized in Issue 77 of Taiko [13]. The key idea is to group constraints into clusters in the size range $\{1, 2, \ldots 2^{k_{ext}-k}\}$ where k_{ext} is the largest degree constraint size, and the subtraction due to k is taking into account the degree reduction due to division of the vanishing polynomial. Each cluster contains a subset of columns that participate in the cluster and each column in a cluster only needs to be extended to the maximum degree of the cluster itself. Each cluster is further subdivided in parts that are executed in sequence. The savings is that fewer columns are extended to higher domains and there will be a net extended savings in overall memory and compute.

This sounds quite promising, except that due to the data dependency across gate, copy and permutations it is not easily parallelizable. A given column might participate in a small degree gate in one cluster, a large degree gate in another cluster, and a permutation in another cluster and so on. Therefore a choice has to be made, as to whether to copy data from one cluster to the other or recompute. The former makes it non parallel, and the latter reduces performance.

The strategy and the challenges are best explained using a simple example.

Consider a simple constraint evaluation where a_0, a_1, a_2, a_3, f_3 are various trace columns and r is a random parameter that separates mathematically independent constraints. For example let us take the composition polynomial to be

$$\Phi = (a_2 + a_1^2) + r \cdot (a_1 \cdot f_3 \cdot a_3) + r^2 \cdot (a_0 + a_2 \cdot f_3) \tag{3.1}$$

The idea is to separate the constraint expression into clusters of terms of different degrees

$$C_0^n = \{a_0, a_2\} \ , \ C_1^{2n} = \{a_1^2, a_2 \cdot f_3\} \ , \ C_2^{4n} : \{a_1 \cdot a_3 \cdot f_3\} \tag{3.2}$$

where the degree $3n$ cluster has been extended to degree $4n$ for efficient NTT.

The methodology to compute the cosets and evaluate each of these clusters is as shown in fig. 6. Each of the coset parts can be computed in parallel. The main problem as we indicated in the figure is the interdependence of the clusters. Since columns participate across multiple clusters in general, the extension in a cluster may be reused or recomputed for the higher degree cluster. In the former case it affects the parallelizability of clusters and in the latter case it increases latency by repeating coset computation.

In the following section §4, we present some of our research into efficient evaluation of expressions and community detection for parallelizing the quotient polynomial computation.

4 Graph Methods

In this section, we present some of our work in parallelizing quotient polynomial computation inspired by graph approaches as originally suggested by Zcash [14]. For all circuits,

[11] https://github.com/zcash/halo2/issues/427
[12] https://github.com/scroll-tech/halo2/pull/28
[13] https://github.com/taikoxyz/zkevm-circuits/issues/77
[14] https://github.com/zcash/halo2/issues/427

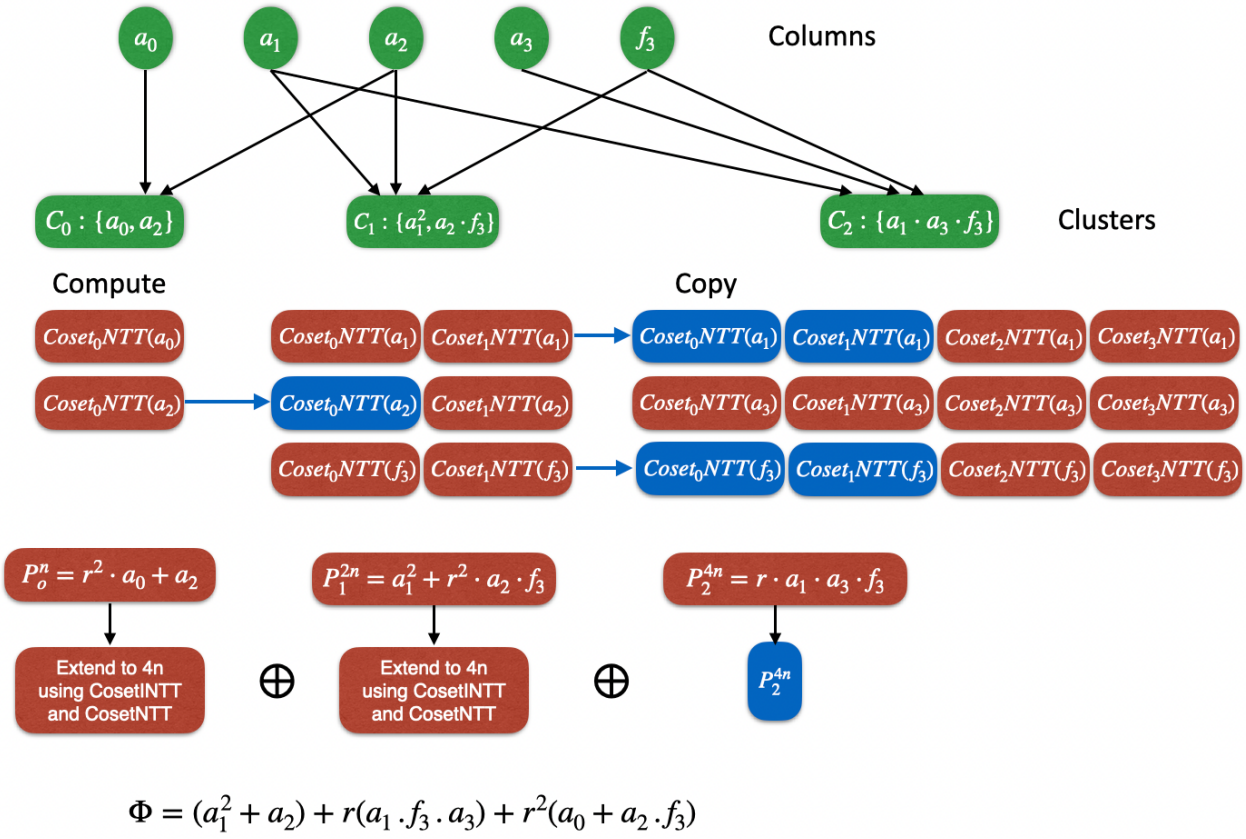

Figure 6. h-poly evaluated using the cluster method, the red blobs are computations (NTT) and can be parallelized, the blue blobs indicate evaluations that need to be stored to be used later. One can also decide to skip the copy and re-evaluate depending on situation, though it is not so easy to implement in a generic way

the symbolic constraint expression forms are either hardcoded or generated by a DSL (Domain Specific Language). In Halo2 this information is either accessible from the Prover key/verifier Key or at the time of evaluation of the constraints. A relationship graph can be built while parsing the above steps. The type of graph constructed can depend on various analysis parameters. We begin by a few definitions that will help us get oriented. For the purposes of this discussion, we only consider un-directed graphs.

Definition A simple graph G is a pair (V, E) where V is a finite set and E is a set of all 2 element subsets of V [12]. The set $V : V(G)$ is the set of vertices (nodes) in G. For $v_1, v_2 \in V$ that are two distinct elements, if there exists a connection between the nodes v_1 and v_2 then $v_1 - v_2 \in E(G)$ is called an Edge set.

For example let G be a simple graph defined by

$$G : \{\{v_1, v_2, v_3, v_4\}, \{v_1 - v_3\}, \{v_1 - v_4\}, \{v_3 - v_4\}\} \tag{4.1}$$

Then, the vertex edge sets are $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{v_1 - v_3, v_1 - v_4, v_3 - v_4\}$.

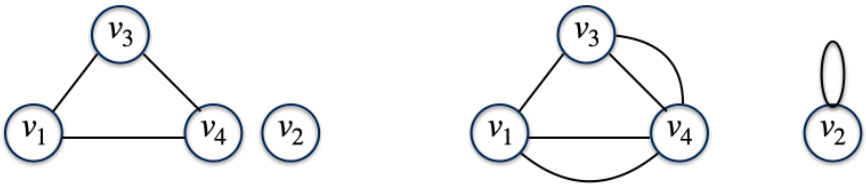

Figure 7. On the left side is a simple graph, on the right side is a multi graph that can be collapsed to a simple graph

Note also that there is no edge to the node v_2 and one can define a graph as a union of several disjoint graphs as well.

Definition A multi graph G is a triple (V, E, ϕ) where V, E are two finite sets and $\phi \rightarrow \mathcal{P}_{1,2}$ is the set of all one element or two element subsets of V. A multi graph can include loops (see fig. 7), whereas a simple graph does not. All multi graphs are collapsible into simple graphs. See for eg $\{v_1, v_2, v_3, v_4\}$ and $E(G) = \{v_1 - v_3, v_1 - v_4, v_3 - v_4, v_4 - v_1, v_4 - v_3, v_2 - v_2\}$ in fig. 7 can be collapsed to (4.1). However, the collapsing process removes loops and parallel edges are collapsed into a single edge and is an irreversible process w.r.t to the given graph. Expression evaluation given a set of columns v_i can be given a graph representation with a definition or heuristic to build the graph. In the next section we discuss a graph primitive that is very helpful in analyzing graphs known as connected components analysis.

4.1 Connected Components

A Connected Components Analysis (CCA) is an approach where given a graph G, subsets of connected components can be labelled using heuristics. An example of such a heuristic is the Layouter in halo2, this creates a dot graph representation of the circuit and uses a heuristic called region. The region is a group of columns and rows with filled cells that preserve the relative positions of the cells. Given a graph representation of a halo2 circuit, a region is a heuristic that defines a sub-graph in the circuit which is a connected component.

We wish to apply a similar heuristic in evaluation, i.e to find a way to classify terms in an expression based on connected components as a principle. Let $R(v_i)\ \forall v_i \in M_{n\times m}$ where M is the trace table and v_i the columns, be the symbolic representation of constraints in a composite polynomial (2.9). We define the connected component heuristic following the ideas suggested by ZCash [15] as

- Each column $v_i \in M$ is labelled as a variable and is assigned to the vertex set V.
- Monomials s_i are functions of variables v_i that are also assigned to the vertex set V.
- Each monomial s_i that is connected to all the variable vertices $\{v_i\}$ gets an edge to the corresponding vertices that participate in this monomial. The edges are represented as $\{s_i - v_1, s_i - v_2, \ldots\}$

[15] https://github.com/zcash/halo2/issues/427

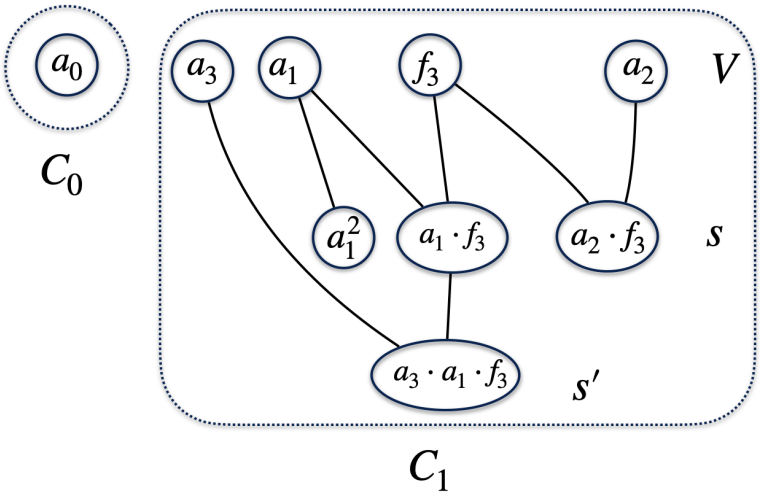

Figure 8. Connected components analysis, the dotted boxes indicate disjoint connected components. Each column and monomial gets its own vertex in this representation.

- Each monomial can further be connected to super-monomial vertices that have this monomial as a subset.
- Since we represent monomials as vertices, it can take into account loops in the graph.
- In halo2, columns have rotations, which can be taken into account during evaluation by shifting the offset appropriately.

For example, the constraint from §3.2

$$\phi_1 = (a_1^2 + a_2) + r \cdot (a_1 \cdot a_3 \cdot f_3) + r^2(a_0 + a_2 \cdot f_3) \tag{4.2}$$

in graph representation. Let $V = \{a_0, a_1, a_2, a_3, f_3, s\}$ be the column vertices and the monomial vertices be $s \subset V = \{a_1^2, a_1 \cdot f_3, a_2 \cdot f_3, s'\}$, where we have defined another monomial vertex $s' \subset s$ with $s' = \{a_1 \cdot a_3 \cdot f_3\}$. Indexing each set V, s, s' with $i = 0, 1 \ldots$, the edges are given by

$$E = \{V_1 - s_0, V_1 - s_1, V_2 - s_3, V_3 - s_0', V_4 - s_1, V_4 - s_2, s_1 - s_0'\} \tag{4.3}$$

with connected components

$$C_0 = \{V_0\} \ , \ C_1 = \{V_1, V_2, V_3, V_4, s_0, s_1, s_2, s_0'\}, \tag{4.4}$$

$\{C_0, C_1\}$ with degrees $\{1, 3\}$ respectively and number of components $|C_0| = 1, |C_1| = 8$ as shown in fig. 8. We make the following observations

- Each element in a connected component set can be extended to the maximum degree of the connected component and only extended further when required for recombination to evaluate the full constraint.

- The connected components of the graph corresponds to different maximal sizes of NTT required for the vertices v_i in each connected component.
- When there are many disjoint connected components, each connected component can be dealt with independently in parallel. Though, this occurs less frequently in practice.
- Only the evaluations of the polynomials corresponding to the variables in a connected component are required to be stored in memory for computation.
- Connected components of size 1 do not need to be extended to a larger domain until necessary.

The pseudo code for connected components is presented in Algorithm 1. The output of the algorithm is

- A mapping of the columns v_i to all the monomials in G
- Connected components $B(\{bin\})$
- For each $bin \in B$, the metadata: maximum degree of that bin, all sets of columns participating in the bin, and an index map of all the monomials s to a indexed constraint list.

Once the connected components are sorted out in bins, the quotient polynomial can be evaluated following the pseudocode in algorithm 2. This way one can save a large number of unnecessary NTTs. The main caveat of this method was already seen in §3.2, when the graph becomes larger, permutations and lookups, (especially permutations) can render the whole graph into one massive single connected component §A. Nevertheless, we believe there is further research that can be done perhaps to use the connected component idea as a heuristic to design efficient circuits or to use as a sub protocol for efficient evaluation.

When we tried the connected components algorithm on a version of PSE super circuit [16], we obtained four groups of connected components.

$$|C_0| = 499 \ , \ |C_1| = 1 \ , \ |C_2| = 52 \ , \ |C_3| = 3 \tag{4.5}$$

which looks very difficult to parallelize, and is likely to face potential bottlenecks in memory, since one connected component is really large compared to the rest. The very specific limitation is essentially due to the permutations and lookups, and this forced us to rethink the problem in a different direction, which is to cluster them into communities, with as weak connectivity as possible and not as strictly disjoint connected components. We will deep dive into that in the following section.

4.2 Community detection

In situations where an entire graph is connected, then the first part of algorithm (1) will output a single connected component. In this case, we formulate the problem as follows.

Algorithm 1 Connected components

input $R(v_i) \forall\ v_i \in M$, indexed constraint list $C \equiv \{\phi_i\}$ such that $\Phi = \sum_i \phi_i \cdot r^i$
Construct graph: $G = (V_1, V_2, E)$
$V_1 \leftarrow v_i\ \forall i = 0, \ldots m-1$ ▷ All columns
$V_2 \leftarrow s_i$ ▷ monomial subexpressions $\{s_1, s_2 \ldots\} \in \Phi$
for $j = 0, 1, \ldots |V_2| - 1$ **do**
    for $i = 0, 1, \ldots m-1$ **do**
        if $v_i \in s_j$ **then**
            $E \leftarrow (v_i - s_j)$ ▷ Edge relations
        end if
    end for
end for
Connected components bins: $B(\{bins\}) \leftarrow BFS(G)$ ▷ Breadth First Search
map connected components to constraints metadata
for $bin \in B$ **do**
    $\text{MD}_{bin} \leftarrow maxdegree(\{s\} \in bin)$ ▷ Max Degree
    $\text{CV}_{bin} \leftarrow v_i \in bin$ ▷ Column Variables
    for $s \in bin$ **do**
        for $\phi \in C$ **do**
            if $s \in \phi$ **then** Perhaps w
                Map $\text{MC}_{bin} \leftarrow \{s, \phi[index]\}$ ▷ Monomial to constraints map
            end if
        end for
    end for
end for
output:$ColumnToMonomials \leftarrow E$
output:$B(bin), BD \leftarrow \{bin, \text{MD}_{bin}, \text{CV}_{bin}, \text{MC}_{bin}\}$ ▷ Bin to data map

Given a graph G is it possible to partition the graphs into groups such that the inter group edges are sparse, while the intra-group edges are dense. In other words, is it possible to partition a graph into k groups (see fig. 9) such that the number of edges between the groups is minimum [13]. Ideally, we would want to have an algorithm that gets as input the graph G and a target number k, and outputs the group partition of the graph to target number sets, such that all of the sets are roughly "similar" in size and have "small" intersection. However, this turns out to be quite challenging, since the graph partition problem is a known NP hard problem and we can best hope for heuristic solutions.

The definition of a community is not well defined - most of the definition of communities are based on heuristics (such as minimizing the modularity of the partition or partitioning according to the "edge-betweenness" of edges). It is unclear how to formulate the notion of a partition for "large" communities with a "small" number of crossing edges between the

[16] https://github.com/privacy-scaling-explorations/zkevm-circuits/blob/main/circuit-benchmarks/src/super_circuit.rs

Algorithm 2 Evaluating quotient polynomial with connected components

Input: Input for quotient polynomial, B, BD from algorithm (1)
Input: Indexed constraint list $C \equiv \{\phi_i\}$ such that $\Phi = \sum_i \phi_i \cdot r^i$
Init: $CP_{coeff} = [0]_{n \cdot d}$ ▷ d : max degree of constraint
for $bin \in B(bin)$ **do**
    Init: $d_{bin} = n \cdot MD_{bin}$ ▷ Max Degree of bin
    Init: $cp_{bin} = [0]_{d_{bin}}$
    In place $CosetNTT_{d_{bin}}(v_i)\ \forall\ v_i \in CV_{bin}$ ▷ This step can be further optimized
    for $s \in MC_{bin}$ **do** ▷ Monomials vertex in bin
        $t \leftarrow$ **Eval**(s, v_i) ▷ Adds, mults including rotations
        for $i = 0, \ldots |C| - 1$ **do** ▷ To get the correct powers of r
            if $s \in MC_{bin}(s, \phi[i])$ **then**
                $cp_{bin}[i] += t \cdot r^i$ ▷ r from $\sum_i \phi_i r^i$
            end if
        end for
    end for
    In place $CosetINTT_{d_{bin}}(cp_{bin})$ ▷ Coefficient form
    $CP_{coeff} += [cp_{bin} || 0_{n \cdot d - d_{bin}}]$
end for
Output: $h_{poly} = CosetNTT_{n \cdot d}(CP_{coeff})/V_H$ ▷ V_H is the vanishing polynomial

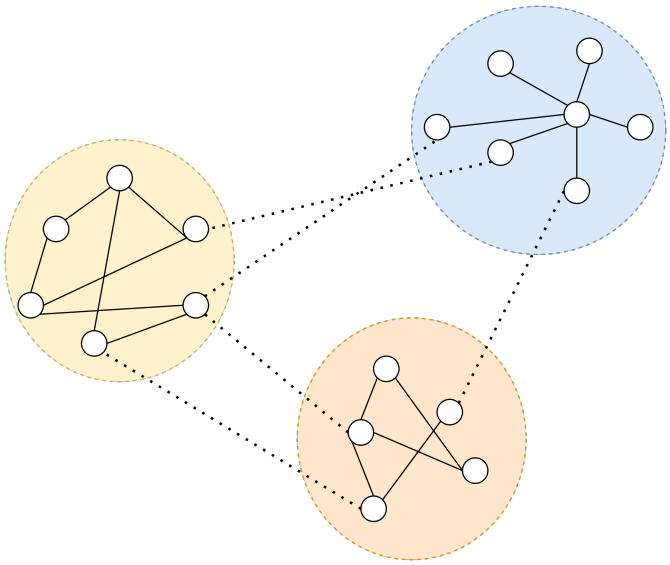

Figure 9. Graph partition: communities. The dotted lines represent the edges between the groups, while the colored groups are densely connected.

communities. In addition, we do not have any idea of how the graph should look like, and hence cannot make assumptions on its properties. In particular, we cannot favor or rule out community detection heuristics. If we choose to apply some community detection algorithm,

we may get a list of communities, but the question then would be how to process these sets to get the bin partition. Even ignoring the requirement that the intersection needs to be small, the requirement that the sets need to have similar size, makes this problem quite similar to the multi-way number partitioning which is also a known NP-hard problem [14].

Some pointers towards the difficulty of satisfying both requirements are presented here [15], which we leave for future research. In general the following greedy algorithm could lead to a quick result.

- Run some community detection algorithm. In our approach we attempt to partition with one of the most well known algorithms that can be used to evaluate graph partition quality known as Girvan-Newman Algorithm [8] which we will cover in the §4.3.
- Merge the communities in a greedy way to get them to be of "similar" size (this could be done in several greedy strategies).
- Create the bin partition (i.e., identify the columns that participate in more than one bin and copy these columns to both partitions).
- The goal is then to run evaluation corresponding to each bin in a different device such as a GPU, which enhances parallelization and also solves memory bottlenecks.

In the following section, we briefly describe the theory behind the Girvan-Newman algorithm and present a method for clustering Taiko's ZKEVM super circuit [17].

4.3 Girvan-Newman Algorithm

The key idea in identifying communities is to compute edge betweenness centrality [8] of all the edges. It is formally defined as follows. Given a graph $G : (V, E)$ edge betweenness of a given edge $e \in E$ is the number of shortest paths between pairs of vertices $\{v_1, v_2\} \in V$ that contain e, normalized by the total number of shortest paths that run between $\{v_1, v_2\} \in V$.

$$EB(e \in E(G)) = \sum_{v_1, v_2 \in V} \frac{\sigma_{v_1,v_2}(e)}{\sigma_{v_1,v_2}} \tag{4.6}$$

If the graph contains groups with few inter group edges, then all the shortest paths between the groups must pass through the few inter group edges. Thus, the edges connecting different communities must have high $EB(e)$. In order to reveal the community structure, one can iterate by removing the edges with high edge betweenness. Let us consider a simple example as shown in fig. 10, the number of shortest edges that pass through $e = C - D$ are

$$EB(C-D) = 9\ ,\ \mathcal{P}(C-D) = \{A-D, A-E, A-F, B-D, B-E, B-F, C-E, C-F, C-D\} \tag{4.7}$$

On the right hand side of fig. 10 once we removed the edge $C - D$ the edge betweenness of the other edges decreased, but the cluster becomes more tight and the number of connected

[17] https://github.com/taikoxyz/zkevm-circuits/tree/fix-super

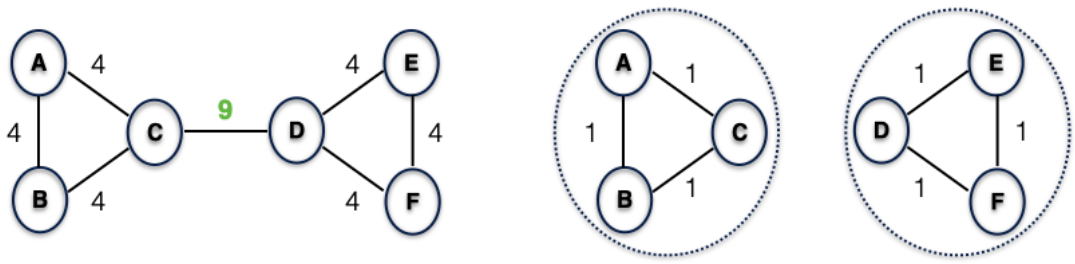

Figure 10. Girvan-Newman: The edges that connect between communities have high betweenness, repeatedly peeling of high betweenness edges reveals community structure hierarchically

components in the edged removed graph set become larger. This iterative process can be continued subject to different heuristics, and in our case we are essentially interested in isolating a few large approximately equi-sized communities, for partitioning the evaluations. Further iterations would result in a community structure like that of a dendrogram (a tree like structure), with communities, sub communities and so on, which is less interesting from a global partitioning point of view. [18]

Once we got the connected components and the largest community, we can convert this into a practical method using the preprocessing method. (This is not unique and can also be done depending on how the h-poly code is written)

1. Build a graph $G = (V, E)$, where all the columns $v_i \in V$ and all the edges $e \in E$. If e_1 and e_2 follow one of the following conditions

   - e_1 and e_2 participate in the same custom gate.
   - e_1 and e_2 are in the same permutation set.
   - e_1 and e_2 are used in the same lookup argument.

2. Apply Algorithm (3) to graph G , get communities $S = s_0, \ldots s_n$ and shared edges E_{copy} across communities.

3. Sort the communities by their size descending. Take the biggest community as the first bin b_1.

4. Merge the communities $s_1 \ldots s_n$ to get the columns needed for the second bin and so on. In the case of our POC, we have chosen to have 2 bins, to get a split across two devices (targeting a potential 2 GPU parallel solution)

[18] We provide a rust implementation of the algorithm (3) in https://github.com/ChickenLover/petgraph/blob/master/src/algo/community.rs

Algorithm 3 Girvan-Newman Algorithm

Input: $G : (V, E), k$ ▷ k is the number of iterations
for $i = 0, 1 \dots k$ **do**
  $CC_{old} = getConnectedComponents(G).len()$ ▷ Compute CC
  $CC_{new} \leftarrow CC_{old}$
  while $CC_{new} = CC_{old}$ **do** ▷ number of CC increases when graph partitions
  for $e \in E$ **do** ▷ Compute Edge Betweenness
    $t.append(EB(e))$
  end for
  $MB \leftarrow Max(t)$ ▷ Compute Max Betweenness
  for $e \in E$ **do**
    if $EB(e) = MB$ **then**
      $ER.append(e)$ ▷ Edges to be removed with Max MB
    end if
  end for
  for $er \in ER$ **do**
    $G \leftarrow RemoveEdge(G, er)$ ▷ Remove Edges
  end for
  $CC_{new} \leftarrow getConnectedComponents(G).len()$ ▷ Compute CC
  end while
end for
Output: $communities \leftarrow getConnectedComponents(G)$, ER

5. For each $e \in E$ check if $e_1 \in b_1$ and $e_2 \in b_2$ (or vice versa). If this is the case $b_2 \leftarrow e_1$ (or $b_1 \leftarrow e_2$). The number of copied columns across the bins determines the computational overhead created by using the community approach.

6. For each bin b_i create boolean vectors G, P and L. The i'th element of G, P and L would be set to 1 if the gate indexed i, permutation set i or lookup argument indexed i respectively would need to be evaluated in the context of the bin b_i.

7. This algorithm was implemented in Rust for the CPU and is intended to run on the CPU. The result of the algorithm is a set of bins $B = b_i(c_i, g_i, p_i, l_i)$, where

   - $C = c_i$ Columns used in this bin,
   - $G = g_i$ Custom gates that need to be evaluated,
   - $P = p_i$ Permutation sets that need to be evaluated,l
   - $L = l_i$ Lookup arguments that need to be evaluated.

which can be then used to evaluate the quotient polynomial.

4.4 Application to Taiko supercircuit

In this section we present our results on parallelization of some of the ZKEVM super circuits. Application of the Girvan-Newman algorithm on the Taiko super circuit [19] for $(k = 2)$ in algorithm (3) gives

$$|C_1| = 303 \ , \ |C_2| = 203 \ , \ |C_3| = 46 \tag{4.8}$$

Following our heuristic in the previous section we merge the smaller communities $C_{2,3} = \{C_2, C_3\}$ to create two communities of approximately even initial memory with minimal number of edges passing across them.

$$|C_1| = 303 \ , \ |C_{2,3}| = 254 \ , \ |E_{copy}| = 23 \tag{4.9}$$

Where E_{copy} are the number of shared edges (columns) across the final communities. The number of overlapping edges between C_1 and $C_{2,3}$ is 23 is an acceptable trade-off since the ratio of the larger community C_0 to that of the total circuit size is about 54% and we have achieved a good partition.

Data	**Base level**	**Net:ours**	**Bin 1**	**Bin 2**
Columns	552	557	303	254
Instance	2	2	0	2
Advice	484	489	274	215
Fixed	66	66	29	37
Permutation sets	3	3	1	2
Lookups	244	244	114	130
Total Instance size	20.15 GB	20.51 GB	10.20 GB	10.31 GB
h-poly time	623 secs	826 secs	552 secs	274 secs

Table 2. Parallelization of quotient polynomial using Girvan-Newman algorithm. Circuit degree 2^{19}. In the bin_1 the column count includes the instance

Our implementation runs on top of Taiko's cluster optimization and following a proposed memory optimization [20] we extracted the columns and symbolic expressions at the time of the evaluation. The implementation POC produces a valid proof and passes verification for the super circuit. The results in table 2 were obtained after running our implementation in a $i9 - 12900K$ 16 core CPU with 24 threads. We did not attempt to optimize the graph algorithm, as in general it will be part of the setup, which adds to the net time in the table. The main point is that in a two GPU setup, bin_1 and bin_2 can run completely independent in parallel, thus reducing overall latency significantly.

4.5 Application to Scroll ZKEVM circuit

Following our initial success with the community detection on the Taiko supercircuit, we ran it on the Scroll supercircuit [21] and discovered different results. We find that there is no

[19] https://github.com/taikoxyz/zkevm-circuits/tree/fix-super
[20] https://github.com/taikoxyz/zkevm-circuits/issues/77
[21] https://github.com/scroll-tech/zkevm-circuits.git

good partition, since the size of the larger community is about 68% of the total circuit size, which is much bigger (see table 4 than that of Taiko 2). In table 3 we see that repeated

Circuit	Connected components	Merged communities	Edges to copy
Super circuit $k = 2$	$\|C_1\| = 658, \|C_2\| = 88, \|C_3\| = 26,$ $\|C_5\| = \ldots = \|C_{16}\| = 14$	$\|C_1\| = 658$ $\|C_{2-16}\| = 309$	$\|E_{copy}\| = 19$
Super circuit $k = 3$	$\|C_{1,1}\| = 564, \|C_{1,2}\| = 94, \|C_2\| = 88, \|C_3\| = 26$ $\|C_4\| = 20, \|C_5\| = \ldots = \|C_{16}\| = 14$	$\|C_{11}\| = 564$ $\|C_{1,2-16}\| = 411$	$\|E_{copy}\| = 35$

Table 3. Communities and connected components for $k = 2, 3$

application of the GN algorithm 3 only reveals sub-communities as it should. In fact, the first community C_1 for $k = 2$ was just sub clustered into $C_{1,1}, C_{1,2}$ and this tree like sub-structure will continue on further application. In terms of practicality, we will be forced to work with the large community as a single unit as this minimizes the number of columns copied. As we can see in table 3 breaking up the large community forces more edges to be copied. Which becomes more and more sub optimal upon further global partitioning. At first we were a bit puzzled as to why the Scroll super circuit has a much larger community, we believe it is largely due to the Scroll super circuit having a large number of permutation columns, 190 as opposed to 15 in Taiko super circuit. As we explained in appendix §A permutations increase the number of connected components significantly.

Data	**Super ciruit**	**EVM**	**State**	**keccak**	**byte code**	**pi**	**tx**	**exp**
Columns	971	236	91	112	26	35	211	43
Advice	736	211	84	94	20	20	139	39
Instance	1	0	0	0	0	1	0	0
Fixed	234	25	7	18	6	14	72	4
Lookups	113	20	14	27	2	1	17	14
Permutation columns	**190**	8	1	0	0	12	75	0

Table 4. Circuit stats in Scroll ZKEVM, super circuit

4.6 Visualization of communities in Taiko and Scroll super circuits

With the graph from the circuit structure, it is easy to run it on any graph visualization software such as graphviz [9]. Graph visualization and coloring can in general be a bit inaccurate and to be interpreted with care, but they do indicate a general structure which has its own merit. In fig. 11 and in fig. 12 the different "countries" represent connected components, and the more boundaries a country shares with another country in the graph, less likely it can be separated as an independent community without paying higher costs on copying the shared nodes (columns).

We can clearly see that in fig. 11 the grey country is a big community, that shares its borders with at most two countries, leaf green and yellow. The remaining countries, yellow,

green, red are all sharing boundaries and can be grouped as one big community. This is consistent with our analysis and results from §4.4.

On the other hand in fig. 12 there are several tiny snow-flake like islands floating around. These islands are the small communities of size |14| in table 3 (these are actually the contributions from the tx sub-circuit in the super-circuit). However, the number of countries that share a border with another country is significantly more than in fig. 11, this makes it very hard to partition it efficiently. The high inter-connectedness of the countries, makes it difficult to have a two way or even a 3 way partition without paying a high cost for the shared vertices, which is also consistent with our analysis in §4.5.

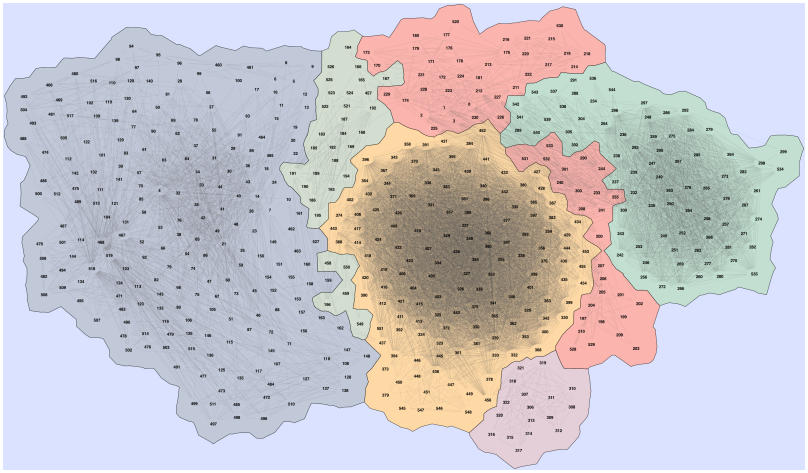

Figure 11. Taiko Super circuit cluster relations highlighted using gvmap clustering in graphviz.

5 Discussion And Future Work

In this work we presented a graph based approach for analysing evaluation of quotient polynomial in halo2. We presented our results in efficient evaluation using connected components and separability problems using community detection. We hope this work inspires more research in this direction and perhaps facilitate early stage research in circuit design with future hardware bottlenecks in mind and development of heuristics that can help achieve this goal.

We leave here some open questions for future research and further consideration. In our algorithm 3 we used a criteria for iteration that as the algorithm should continue to repeat itself until it finds that the number of clustered connected components increase. As we sequentially apply GN, it will keep dividing clusters into sub clusters and so on. One criteria which would terminate the algorithm without specifying it artifically to stop is modularity. It is a criterion to determine where to best cut the groups to get the best clustering. Modularity is defined as

$$Q = \sum_i \left(e_{ii} - \left(\sum_k e_{ik} \right)^2 \right) \tag{5.1}$$

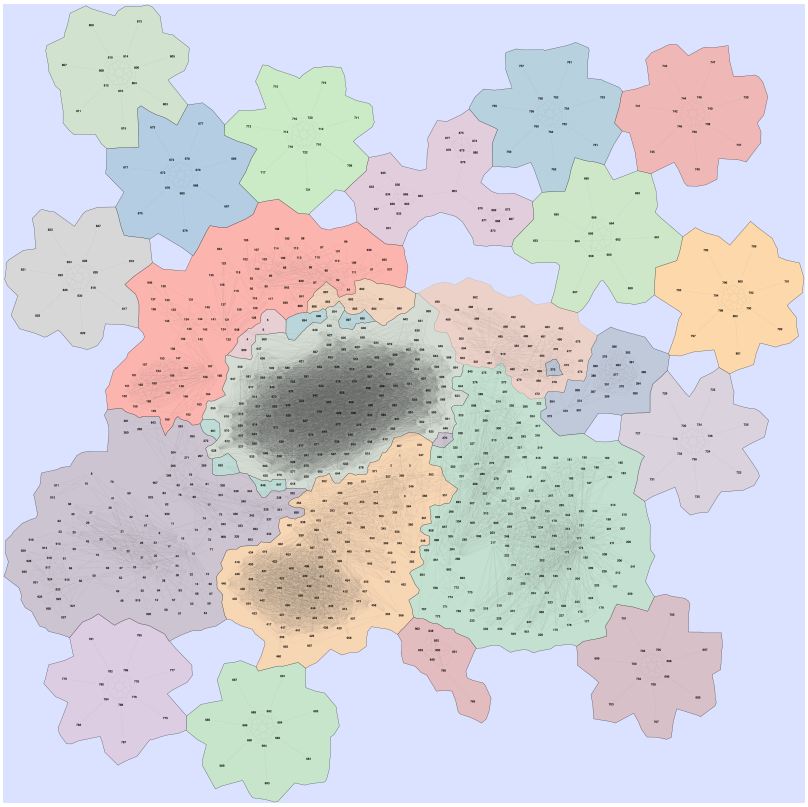

Figure 12. Scroll Super circuit cluster relations highlighted using gvmap clustering in graphviz.

where the summation runs over clusters, the first term $\sum_i e_{ii}$ represents the fraction of edges within a cluster i, and summed over all clusters. The second term $\sum_i (\sum_k e_{ik})^2$ represents the fraction of edges that connect between one cluster to the other. For a random cluster $Q \sim 0$, whereas for a significant community structure $Q \sim 1$. We have not applied this criteria in our exploration, and it would be interesting to take this into account as it gives a metric for good clustering. Moreover the connected component method can in general be used at any part of the evaluation to identify common sub expressions and determine efficient evaluation strategies.

Acknowledgments

First, we would like to thank ZCash - we stand on the shoulders of giants. We also thank Yuval Domb, Tomer Solberg and Omer Shlomovits for discussions, suggestions and for reviewing the article. We thank Brecht from Taiko for helping us with the super-circuit-fix used in our POC.

References

[1] S. Bowe, J. Grigg, and D. Hopwood, "Recursive proof composition without a trusted setup." Cryptology ePrint Archive, Paper 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[2] L. B. Schmidt and R. T. Bjerg, "High assurance specification of the halo2 protocol." https://moneroresearch.info/index.php?action=resource_RESOURCEVIEW_CORE&id=187&browserTabID=.

[3] S. Bowe, Y. T. Lai, D. E. Hopwood, and J. Grigg, "The halo2 zero-knowledge proving system." https://github.com/zcash/halo2.

[4] S. Bowe, Y. T. Lai, D. E. Hopwood, and J. Grigg, "The zcash protocol." https://zcash.github.io/halo2/design/protocol.html.

[5] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner, "Proof-carrying data from accumulation schemes." Cryptology ePrint Archive, Paper 2020/499, 2020. https://eprint.iacr.org/2020/499.

[6] S. Bowe, Y. T. Lai, D. E. Hopwood, and J. Grigg, "Inner product arguments in halo2." https://zcash.github.io/halo2/design/proving-system/comparison.html#bcms20-appendix-a2.

[7] A. Kate, G. M. Zaverucha, and I. Goldberg, *Constant-size commitments to polynomials and their applications*, in *Advances in Cryptology - ASIACRYPT 2010* (M. Abe, ed.), (Berlin, Heidelberg), pp. 177–194, Springer Berlin Heidelberg, 2010.

[8] M. Girvan and M. E. J. Newman, *Community structure in social and biological networks*, *Proceedings of the National Academy of Sciences* **99** (2002), no. 12 7821–7826, [https://www.pnas.org/doi/pdf/10.1073/pnas.122653799].

[9] Graphviz, "Graphviz - open source visualization software." https://graphviz.org.

[10] Y. Domb, "Ntt 201." https://github.com/ingonyama-zk/papers/blob/main/ntt_201_book.pdf.

[11] K. Inbasekar, "Notes on kzg commitments." https://github.com/krakhit/papers_and_talks/blob/main/Cryptography_ZK_FHE/papers_notes/notes/KZG.pdf.

[12] D. Grinberg, *An introduction to graph theory*, 2023.

[13] R. Shamir, "Girvan newman algorithm." http://www.cs.tau.ac.il/~rshamir/abdbm/pres/17/Newman.pdf.

[14] S. Mertens, *The easiest hard problem: Number partitioning*, 2003.

[15] N. Bansal, U. Feige, R. Krauthgamer, K. Makarychev, V. Nagarajan, Joseph, Naor, and R. Schwartz, *Min-max graph partitioning and small set expansion*, 2011.

A Permutation argument and connected components

A permutation is a $1 \to 1$, onto map of a set to itself. In plonkish argument permutations are used to check the mapping of the values of different cells into one another which is then enforced as a constraint after witness generation.

Since not all cells are in general mapped in a permutation set, it follows that not all columns participate in the permutation sets. Let ω be the $n = 2^k$ root of unity, we can define the permutation operation as

$$P(M_{j,i}) \equiv M_{j,i} + \beta \cdot \sigma_i(\omega^j) \tag{A.1}$$

for a random $\beta \leftarrow \mathbb{F}$ where $\sigma_i(\omega^j)$ is defined as

$$\sigma_i(\omega^j) = \begin{cases} \delta^i \cdot \omega^j \ , \ (j,i) \to (j,i) \\ \delta^{i'} \cdot \omega^{j'} \ , \ (j,i) \to (j',i') \end{cases} \tag{A.2}$$

where δ is a 2^s root of unity such that $s \geq k$. The first line is the identity permutation which we refer as $I(M_{i,j})$. Since permutations are composable, for cells $\{M_{i,j}\}$ in a permutation set, and including all possible permutations generates the product

$$\prod_{i=0}^{m-1} \prod_{j=0}^{n-1} P(M_{j,i}) \tag{A.3}$$

Then it follows that, the ratio

$$\prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \left(\frac{I(M_{j,i})}{P(M_{j,i})} \right) = 1 \tag{A.4}$$

only when the values $\{v_i(\omega^j)\}$ are identical across the permutation set in which they participate.

For instance, in fig. 13 the disjoint permutation sets are (green) $\{M_{0,0}, M_{n-1,1}\}$ and (blue) $\{M_{1,m-1}, M_{n-1,0}\}$. Thus the mapping of the values

$$\begin{aligned} v_0(\omega^0) &\equiv v_1(\omega^{n-1}) \\ v_{m-1}(\omega^1) &\equiv v_0(\omega^{n-1}) \end{aligned} \tag{A.5}$$

is a constraint that is enforced by checking the index mapping $(0,0) \to (n-1,1)$ and $(1,m-1) \to (n-1,0)$ on the matrix M. We observe that the ratio of the identity permutation to the colored permutation (A.1) for the configuration fig. 13

$$\frac{(v_0(\omega^0) + \beta\delta^0\omega^0)(v_1(\omega^{n-1}) + \beta\delta^1\omega^{n-1})(v_0(\omega^{n-1}) + \beta\delta^0\omega^{n-1})(v_{m-1}(\omega^1) + \beta\delta^{m-1}\omega^1)}{(v_0(\omega^0) + \beta\delta^1\omega^{n-1})(v_1(\omega^{n-1}) + \beta\delta^0\omega^0)(v_0(\omega^{n-1}) + \beta\delta^{m-1}\omega^1)(v_{m-1}(\omega^1) + \beta\delta^0\omega^{n-1})} \tag{A.6}$$

is unity for any random β iff the conditions (A.5) are met. Following our discussion in (4.1) we see from the above expression that permutations generate all possible product expressions between columns. In the above the connected component set is

$$\{v_0, v_1, v_{m-1}, v_0 \cdot v_1, v_0 \cdot v_{m-1}, v_1 \cdot v_{m-1}, v_0 \cdot v_1 \cdot v_{m-1}\} \tag{A.7}$$

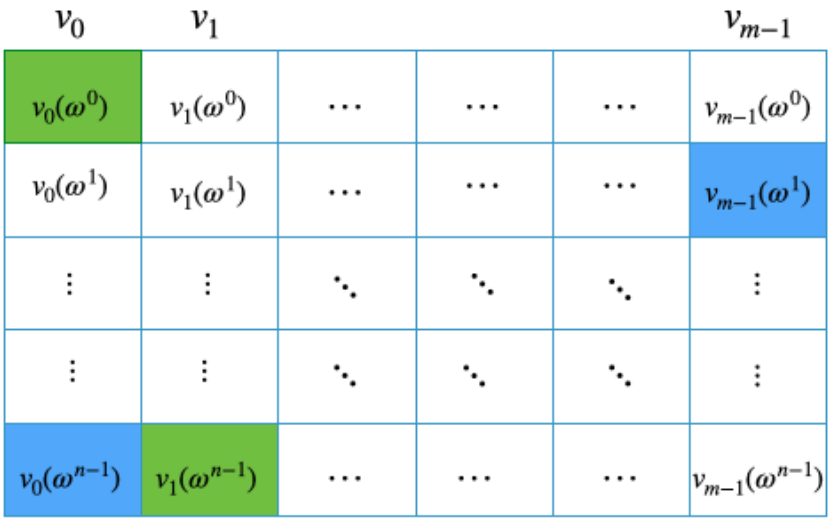

Figure 13. permutation argument: The green and blue cells represent disjoint sets of cell values that are invariant under permutations within each set. In the above we have the permutation sets $\{\{M_{0,0}, M_{n-1,1}\}, \{M_{1,m-1}, M_{n-1,0}\}\}$

Thus permutations tend to clump together columns evaluation when used in quotient polynomial along with gates, and lookups. It is important to note that having many lookups also tend to increase connected component length.