Scalable Globally Optimal Partitioning of Quantum Circuits

Gregory Morse
gregory.morse@live.com

ELTE Eötvös Loránd University, Budapest
Faculty of Informatics
Department of Programming Languages and Compilers

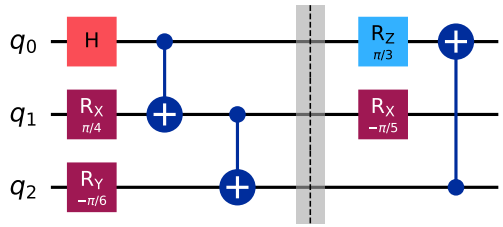
PhD supervisor: Tamás Kozsik

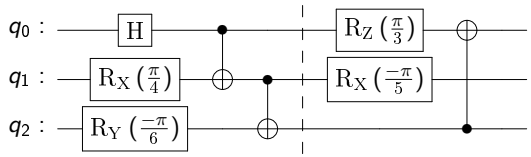
HUN-REN Wigner Research Center for Physics

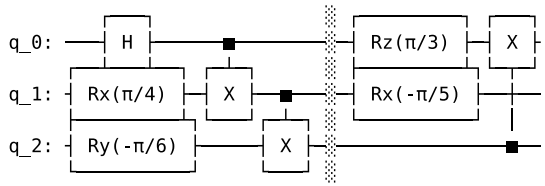
Lectures on Modern Scientific Programming 2025
November 25, 2025

What is a Quantum Circuit?

- A quantum circuit describes a computation using:
  - **Qubits** (quantum bits)
  - **Gates** (operations)
- Gates are arranged in time from left to right.
- Multi-qubit gates create **entanglement**, linking qubit behavior.







A simple quantum circuit acting on 3 qubits drawn with 3 rendering modes of Qiskit: graphical, latex/latex source and text:.

Why Are Circuits Hard to Simulate?

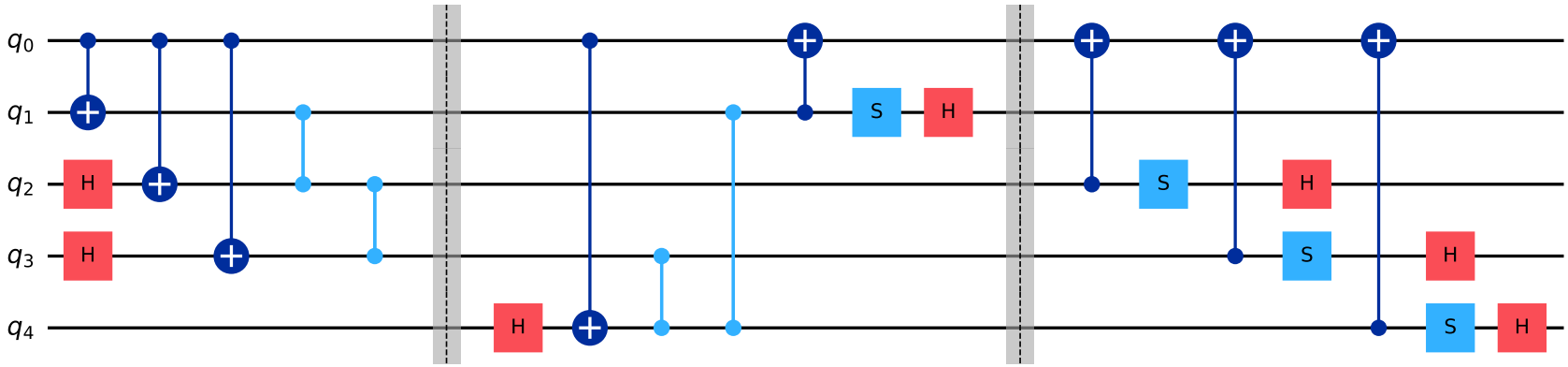
- Each qubit doubles the size of the quantum state.
- A system of n qubits requires 2^n complex numbers.
- Multi-qubit gates act on large blocks of this state.
- Simulation becomes expensive as circuits grow.

Key Idea

If we can break a large circuit into **smaller independent pieces**, we can often simulate or optimize each piece more efficiently.

Partitioning: The High-Level Idea

- Partitioning identifies **subcircuits** that can be:
  - Simulated separately
  - Optimized independently
  - Fused into larger operations
- Helps reduce complexity in:
  - Simulation (CPU/GPU)
  - Decomposition and synthesis
  - Distributed or parallel execution



Example 5-qubit circuit of Perfect Code Encoder into 3 4-qubit Partitions

Where Partitioning Is Used Today

In simulation:

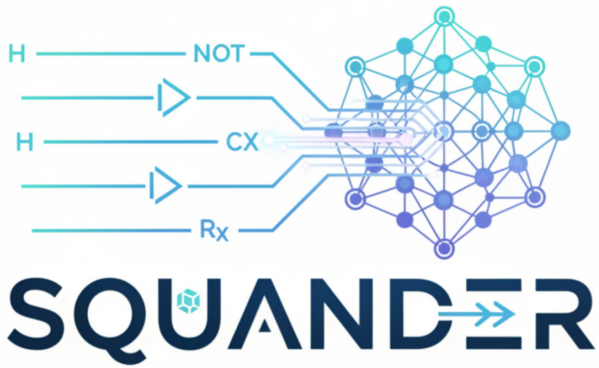
- Find blocks that can be fused into large unitary operations.
- Used in GPU simulators for speed.

In compilation:

- Identify subcircuits to decompose or synthesize.
- Simplify mapping to hardware with limited connectivity.

In distributed computing:

- Split a circuit among multiple processors.
- Reduce expensive communication or teleportation.



Heuristics in Use Today

- **Qiskit:** "Collect multi-qubit blocks" (greedy).
- **BQSKit:** Quick, Scan, Cluster, Greedy.
- **Clark et al. (2023):** Tree-based DAG method.
- **Fang et al. (2022):** Acyclic graph partitioning for simulation.
- **Xu et al. (2024):** ILP assignment of blocks to GPUs (ATLAS).
- **Kaur et al. (2025):** Distributed partitioning over processors.

Observe

These methods work well in practice, but they rely on **heuristics** rather than global optimality.

Limitations of Current Methods

- Hard to know how good a heuristic really is.
- No widely available "gold standard" for optimal partitions.
- Some techniques work well only for certain hardware models.
- Others struggle with:
  - Deep circuits
  - High entanglement
  - Large numbers of qubits

Motivation

We want a method that is **exact** and still **practical**, so we can set benchmarks for the field.

Our Idea in short

- Reformulate partitioning as a **set cover** problem.
- Use modern **ILP solvers** to find the globally optimal solution.
- Add structural ideas to make it scale:
  - Collapse long single-qubit chains
  - Enumerate only **convex**, qubit-aware subcircuits
  - Exclude cycle-forming solutions iteratively
- Result:
  - Optimal partitions for circuits up to **100,000 gates**
  - Solve times on the order of minutes

Why This Matters

- Enables the first "ground truth" dataset for QC partitioning.
- Lets us measure how far heuristics are from optimal.
- Helps guide:
  - Better fusion strategies for simulators
  - Better decomposition strategies for compilers
  - Better cost models for distributed execution
- Provides a foundation for principled quantum compiler design.

Big Picture of the Algorithm

Goal: Compute globally optimal partitions of a quantum circuit DAG.

1. **Preprocess the circuit**
   - Contract simple single-qubit chains.
2. **Enumerate candidate partitions**
   - Use qubit-aware reachability.
   - Enumerate convex subcircuits.
3. **Build an ILP model**
   - Partitions become binary decision variables.
   - Constraints ensure every gate is covered exactly once.
4. **Eliminate cycles between partitions**
   - Ensure the partition graph remains acyclic.
5. **Postprocess**
   - Expand contracted chains and recover the full partitioning.

Pre- and Post-Processing: Single-Qubit Chains

Observation: Long single-qubit chains are easy to handle and do not need to be part of the expensive combinatorial search.

Contraction rules (preprocessing):

- Single-qubit chains with no parents or children:
  - Form trivial partitions and can be removed from the main search.
- Chains with only one neighbor:
  - Always attached to that neighbor's partition.
- Chains with both a parent and a child:
  - In the **unweighted** case, either choice is equivalent.
  - In the **weighted** case, this choice can be NP-hard and is handled by a small ILP.

Postprocessing: After solving the main ILP, the contracted chains are reinserted according to these rules.

Why Qubit-Aware Reachability?

Problem:

- A naive reachability search (e.g., BFS on the DAG) finds many candidate subcircuits.
- Many of these candidates may use too many qubits and cannot be realized by the target hardware.

Idea: Only propagate through parts of the circuit that stay within a given qubit budget $Q_{\max}$.

- Each gate v has an associated set $\mathcal{Q}(v)$ of qubits it acts on.
- While exploring, we keep track of the **union** of qubits visited.
- We stop expanding when this union exceeds $Q_{\max}$.

Result: Infeasible branches are pruned early, and the search space remains manageable.

Circuit DAG and Qubit Sets

- Model the circuit as a DAG:

$$G = (V, E), \quad V = \{\text{gates}\}, \quad E = \{\text{dependencies}\}.$$

- For each gate $v \in V$:

$$\mathcal{Q}(v) \subseteq \mathcal{Q}$$

  is the set of qubits touched by gate v.
- We use:
  - $g(v)$: forward adjacency (successors of v),
  - $rg(v)$: reverse adjacency (predecessors of v).
- Given a starting set of gates X and a qubit bound $Q_{\max}$, we want:

$$\left| \bigcup_{u \preceq v} \mathcal{Q}(u) \right| \leq Q_{\max}$$

  along all paths considered.

Qubit-Aware Reachability: Informal Algorithm

Input:

- Initial gate set X.
- Circuit DAG with adjacency lists g and rg.
- Restriction set R of admissible vertices.
- Mapping $v \mapsto \mathcal{Q}(v)$.
- Qubit limit $Q_{\max}$.

High-level steps:

1. Start from X (current frontier `level`).
2. For each gate v in the frontier:
   - Check whether all required predecessors in R have been visited.
   - If yes, compute the qubit set for v by merging:

$$Q_s[v] = \mathcal{Q}(v) \cup \bigcup_{u \in \mathrm{pred}(v)} Q_s[u].$$

   - If $|Q_s[v]| \leq Q_{\max}$, accept v and add its successors to the next frontier.
3. Repeat until no new gates can be added.

Output: set of gates reachable from X without violating the qubit bound.

Qubit-Aware Reachability: Complexity

Let:

- $n = |V|$: number of gates.
- $k = Q_{\max}$: maximum number of qubits per partition.

Key points:

- Each gate is processed at most once (after predecessors are resolved).
- Qubit sets $Q_s[v]$ are updated incrementally.
- Each update costs $O(k)$ in the worst case.

Overall:

$$\text{Time} = O(n \cdot k), \qquad \text{Space} = O(n \cdot k).$$

In practice:

- k is small compared to n (hardware qubit limit).
- The procedure behaves almost linearly in the circuit size.

What is a Convex Partition?

Convexity in a DAG:

- A subset $P \subseteq V$ is **convex** if:

$$u, v \in P,\ u \preceq w \preceq v \ \Rightarrow\ w \in P.$$

- Intuition: You cannot "skip over" nodes.

Why convex partitions?

- They correspond to **contiguous** subcircuits without holes.
- Easier to interpret, simulate, and decompose.
- Compatible with causal structure of the circuit.

Our goal:

- Enumerate all convex partitions that also respect the qubit bound $Q_{\max}$.

Preprocessing for Enumeration

To speed up enumeration, we precompute:

- A **topological order** $\pi : V \to \{1, \ldots, n\}$.
- For each gate u:

$$\texttt{reach}[u] = \{v \mid u \preceq v\}, \quad \texttt{revreach}[u] = \{v \mid v \preceq u\}.$$

- These reachability sets can be obtained efficiently using standard graph algorithms (e.g., SCC-based methods and dynamic programming).

Benefit:

- These closures allow us to quickly update frontiers when we grow a convex set.
- Combined with qubit-aware pruning, this massively shrinks the search space.

Convex Partition Enumeration: High-Level Strategy

For each seed gate t (in topological order):

1. Start with:

$$X = \{t\}, \quad Y = \{v : \pi(v) > \pi(t)\}, \quad Q = \mathcal{Q}(t).$$

2. Maintain two frontier sets:
   - A: forward-reachable candidates.
   - B: backward-reachable candidates.
3. At each step, choose a frontier node v and compute the region R of nodes that must be included with v to preserve convexity.
4. Extend the current set X by R if the updated qubit set Q' stays within $Q_{\max}$.
5. Use qubit-aware reachability to prune A, B, and Y.
6. When both A and B are empty, record X as a valid convex partition.

Convex Partition Enumeration: Qubit-Aware Pruning

Qubit-aware enhancements:

- Before exploring convex sets rooted at t:
  - Run qubit-aware reachability from t.
  - Remove from Y any node that is not qubit-feasible from t.
- During enumeration:
  - Whenever X grows, we reapply qubit-aware reachability to:
    - prune A and B,
    - drop nodes that would violate the qubit bound.

Effect in practice:

- Huge reduction in the number of candidates we have to explore.
- Enumeration becomes feasible even for large DAGs, as long as $Q_{\max}$ is moderate.

Convex Enumeration: Complexity Discussion

Theoretical worst case:

- The number of convex subsets of a DAG can be exponential.
- Any exact enumeration algorithm can thus be exponential in the worst case.

Our approach:

- Ensures each convex partition is found exactly once.
- Uses polynomial-time operations per step:
  - Updates of reachability frontiers.
  - Qubit-set unions bounded by $Q_{\max}$.

In practice:

- Hardware limits imply small $Q_{\max}$.
- Qubit-aware pruning plus convexity significantly reduce the effective search space.
- This allows us to handle circuits with hundreds or thousands of gates.

What is an ILP?

Integer Linear Program (ILP):

- Optimize a linear objective:

$$\min \quad c^\top x$$

- Subject to linear constraints:

$$Ax \leq b$$

- Some or all variables x_i are required to be integers (often 0–1).

ILPs can model:

- Scheduling, routing, set cover, assignment, and here:
- Selecting an optimal set of partitions.

Modern solvers:

- Highly optimized branch-and-bound / branch-and-cut.
- Exploit structure, cutting planes, heuristics, and presolve.

Mini Example: Set Cover as an ILP

Toy problem:

- Universe of items $U = \{1, 2, 3, 4\}$.
- Subsets:

$$S_1 = \{1, 2\}, \quad S_2 = \{2, 3\}, \quad S_3 = \{3, 4\}.$$

- Goal: pick the fewest subsets so that every item is covered.

ILP formulation:

- Variables: $x_1, x_2, x_3 \in \{0, 1\}$ (choose S_1, S_2, S_3).
- Objective:

$$\min \; x_1 + x_2 + x_3.$$

- Coverage constraints:

$$\begin{aligned} \text{item 1:} \quad & x_1 \geq 1 \\ \text{item 2:} \quad & x_1 + x_2 \geq 1 \\ \text{item 3:} \quad & x_2 + x_3 \geq 1 \\ \text{item 4:} \quad & x_3 \geq 1 \end{aligned}$$

We will use exactly this pattern for gates and partitions.

From Partitions to an ILP Model

Input to the ILP:

- Set of gates $\mathcal{G}$.
- Set of candidate partitions $\mathcal{P} = \{P_1, \ldots, P_m\}$ from convex enumeration.
- For each gate $g \in \mathcal{G}$:

$$\mathcal{P}(g) = \{j \mid g \in P_j\},$$

the indices of partitions that contain gate g.

Decision variables:

$$x_j = \begin{cases} 1, & \text{if partition } P_j \text{ is selected,} \\ 0, & \text{otherwise.} \end{cases}$$

ILP Formulation: Objective and Constraints

Objective:

$$\min \quad \sum_{j=1}^{m} x_j$$

- Minimize the total number of selected partitions.
- This encourages larger, more informative blocks (subject to constraints).

Coverage constraints:

$$\sum_{j \in \mathcal{P}(g)} x_j = 1 \quad \forall g \in \mathcal{G}.$$

- Each gate must be covered by exactly one partition.
- Overlapping partitions are allowed in enumeration, but:
  - The ILP will choose a non-overlapping selection.

Integrality:

$$x_j \in \{0, 1\}, \quad j = 1, \ldots, m.$$

Interpretation and Properties

Interpretation as set cover:

- Gates = items that must be covered.
- Partitions = candidate subsets.
- ILP chooses a minimal collection of partitions that exactly covers all gates.

Why is this globally optimal?

- Every feasible ILP solution corresponds to a valid partitioning.
- The objective ensures we pick the partitioning with the smallest number of blocks, given the candidate library.
- Since we enumerate *all* relevant convex partitions (under the qubit bound), this is optimal within that space.

Computationally:

- The ILP can be large, but modern solvers handle these structures well, especially with the additional structure we add next (acyclicity).

Why Worry About Cycles Between Partitions?

Partition–interaction digraph:

- Nodes represent selected partitions S_i.
- Directed edge $S_i \to S_j$ exists if there is a gate-level edge $u \to v$ with $u \in S_i$ and $v \in S_j$ (exclusive parts).

Potential problem:

- It is possible to select partitions that introduce a cycle in this interaction graph:

$$S_{i_1} \to S_{i_2} \to \cdots \to S_{i_k} \to S_{i_1}.$$

- Such a cycle violates the DAG structure at the partition level and breaks the idea of a hierarchical decomposition.

Goal:

- Enforce that the selected partitions themselves form a DAG.

Canonical 2-Cycle Between Partitions

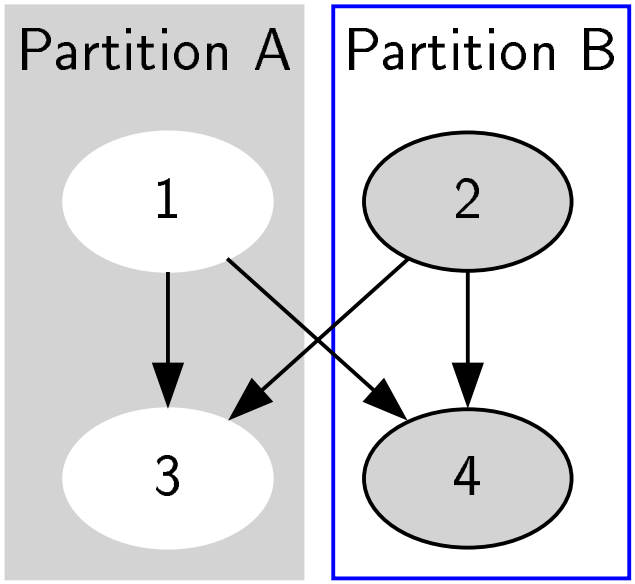


Figure: Canonical invalid 2-cycle between two convex partitions S_i and S_j.

Setup:

- S_i and S_j are both individually valid convex partitions.
- There is a cross-edge $S_i \to S_j$: an edge $u \to v$ with $u \in S_i \setminus S_j$, $v \in S_j \setminus S_i$.
- There is also a cross-edge $S_j \to S_i$: an edge $u' \to v'$ with $u' \in S_j \setminus S_i$, $v' \in S_i \setminus S_j$.

Consequence:

- If we select both S_i and S_j, the partition–interaction graph contains a directed 2-cycle.
- This violates the global acyclicity requirement.

Cycle cut:

- $x_i + x_j \leq 1$
- At most one of S_i or S_j can be chosen.
- This is the simplest non-trivial example of our general cycle cuts.

Strategy 1: Cycle-Elimination Cuts

Idea: Detect cycles in the partition–interaction graph and forbid them with linear inequalities.

For a directed cycle C:

$$\sum_{i \in C} x_i \leq |C| - 1.$$

- At least one partition in the cycle must be switched off.

Special cases:

- 2-cycle: $S_i \leftrightarrow S_j$:

$$x_i + x_j \leq 1.$$

- 3-cycle: $S_i \to S_j \to S_k \to S_i$:

$$x_i + x_j + x_k \leq 2.$$

Works well when:

- The number of short cycles is moderate.
- Used in combination with other strategies.

Strategy 2: Order Variables (Topological Labels)

Idea: Assign each partition a real-valued **order** u_i that respects the direction of edges.

Constraints:

- For each edge (i, j) between partitions:

$$u_j \geq u_i + 1 \quad \text{if } x_i = x_j = 1.$$

- If either x_i or x_j is 0, the constraint can be relaxed (via a big-M term or an indicator constraint in the solver).

Why this prevents cycles:

- Summing along a directed cycle would give:

$$u_{i_1} \geq u_{i_1} + |C|,$$

  which is impossible.
- Therefore, no set of selected partitions can form a cycle.

Advantage: Single, compact ILP model; no need to detect cycles explicitly during the solve.

Strategy 3: Lazy Cycle Cuts (Cutting-Plane Approach)

Idea: Let the solver propose a candidate solution, then check it for cycles and cut them off if they exist.

Procedure:

1. Solve the ILP *without* acyclicity constraints.
2. Inspect the selected partitions ($x_i = 1$) and build their interaction graph.
3. If there is a cycle:
   - Extract the cycle C.
   - Add the cut:
   $$\sum_{i \in C} x_i \leq |C| - 1.$$
   - Reoptimize and repeat.
4. If there is no cycle, we have a valid solution.

Modern MILP solvers:

- Support this via **lazy-constraint callbacks**.
- Often very efficient in practice (few iterations).

Cycle Elimination Algorithm: Summary

High-level algorithm:

1. Start with the base ILP formulation (no cycle constraints).
2. Repeatedly:
   - Solve the ILP.
   - Build the partition–dependency graph for the selected partitions.
   - Compute the strongly connected components (SCCs).
   - For each SCC of size > 1, extract a cycle and add a cut.
3. Stop when the selected partitions form an acyclic graph.

Complexity:

- ILP solve dominates the runtime.
- Graph operations (SCCs, cycle extraction) are linear in the number of selected partitions and arcs.
- Empirically, only a few refinement rounds are needed.

Full Pipeline Recap

1. **Preprocessing**
   - Contract single-qubit chains.
2. **Qubit-Aware Convex Partition Enumeration**
   - Use qubit-aware reachability and convexity.
3. **ILP Construction**
   - Partitions $\rightarrow$ binary variables.
   - Gates $\rightarrow$ coverage constraints.
   - Objective: minimize number of partitions.
4. **Acyclicity Enforcement**
   - Cycle cuts, order variables, or lazy constraints.
5. **Postprocessing**
   - Expand contracted chains.
   - Recover full partition mapping for the original circuit.

Result: Globally optimal, hardware-aware partitions for circuits with up to $\sim 10^5$ gates, computed in practical time.

Connectedness of Partitions and Convex Sets

Key Principle: All partition blocks are assumed to be connected subcircuits.
Why only connected convex sets?

- Our convex-set enumeration algorithm was explicitly the connected version: every enumerated block forms a single qubit-connected region.
- Disconnected convex sets offer no advantage:
  - They cannot be fused efficiently in simulation.
  - They provide no benefit for decomposition.
  - They inflate the search space without adding useful partitions.

Partitioning rule (crucial):

- Every candidate block is pre-split into its connected components, regardless of how it was generated.
- Ensures a fair comparison between methods and a consistent basis for the ILP.
- Guarantees that all blocks correspond to meaningful quantum operations on a contiguous set of qubits.

Consequence: The ILP only receives connected, convex, and simulation-relevant blocks, aligning the optimization with real fusion efficiency.

Why We Need a Weighted Objective

Unweighted optimal partitioning:

- Minimizes the number of blocks.
- Ensures convexity and hardware feasibility.

But for simulation or fusion:

- Different blocks cost different amounts of work.
- A block acting on k qubits requires operations on a 2^k-dimensional subspace.
- Larger blocks are exponentially more expensive.

Therefore

We need to attach a **weight** (cost) to each candidate partition:

$$\text{weight}(P_j) = \text{estimated FLOPs to simulate block } P_j.$$

Weighted ILP Objective

Recall the unweighted objective: $\min \sum_j x_j$

Weighted version for gate fusion: $\min \quad \sum_{j=1}^{m} w_j x_j$ where:

- $x_j \in \{0, 1\}$ selects partition P_j,
- w_j is the cost (FLOPs) required to simulate P_j.

Coverage constraints remain unchanged:

$$\sum_{j:g \in P_j} x_j = 1 \quad \text{for each gate } g.$$

Interpretation

The ILP simultaneously:

- chooses partitions,
- avoids cycles,
- respects the qubit bound,
- and minimizes the true floating-point simulation cost.

FLOPs for Applying a Gate Block

Consider a fused block acting on a set of k qubits.

State-vector size:

vector has size 2^n where n is the total number of qubits.

When applying a k-qubit operation:

we update 2^{n-k} disjoint subvectors,

each of length:

$$2^k.$$

Fusion cost intuition

$$\text{Cost} = 2^{n-k} \times (\text{cost to multiply a } 2^k \times 2^k \text{ matrix})$$

Cost of a $2^k \times 2^k$ Complex Matrix Multiply

Each complex multiply: $(a+bi)(c+di) = (ac-bd)+(ad+bc)i$ requires:

$$4 \text{ real multiplications} + 2 \text{ real additions} = 6 \text{ FLOPs}.$$

A matrix–vector multiply of size 2^k has:

$$2^k \text{ outputs}, \quad \text{each requiring } 2^k \text{ complex multiplies}.$$

So the total is:

$$2^k \times (2^k) \times 6 = 6 \cdot 2^{2k}.$$

But SQUANDER uses a slightly richer estimate

including:

- matrix–vector core cost,
- accumulation cost,
- an I/O penalty.

The I/O Penalty

Motivation:

- Loading and storing slices of the state vector is expensive.
- Memory movement, especially across cache lines, dominates runtime.

Approximated as a constant:

$$\texttt{io\_penalty} = 48 \quad \text{(empirical estimate)}$$

It is added once per matrix–vector multiply block:

$$\text{cost} = \text{FLOPs}_{\text{MV}} + \texttt{io\_penalty}.$$

Why this matters

For small blocks (2–4 qubits), I/O dominates; for large blocks, arithmetic dominates.

Putting It All Together: Cost of One Block

For a fused block acting on qubits:

$$\text{gate\_qubits} = k, \quad \text{control qubits} = c,$$

effective size:

$$g_{\text{size}} = 2^{k-c}.$$

The FLOP estimate implemented is:

$$\text{FLOPs}(k, c) = 2^{\,n-(c \text{ if pure else } 0)} \times (g_{\text{size}} \cdot (4+2) + 2(g_{\text{size}} - 1) + \texttt{io\_penalty})$$

- The first factor counts how many slices of the state vector must be updated.
- The second factor is the per-slice computation (matrix–vector multiply + I/O).

Computing the Weight of a Partition

For a candidate partition P_j (a fused block):

$$\text{weight}(P_j) = \text{FLOPs needed to apply } P_j.$$

Steps:

1. Collect the qubits involved in P_j:

$$Q(P_j) = \bigcup_{g \in P_j} \mathcal{Q}(g).$$

2. Determine number of:
   - gate qubits,
   - control qubits,
   - (optionally) whether the block is "pure".
3. Apply the FLOP model per block.

Key idea

The ILP is now minimizing the true simulation cost, not simply the number of blocks.

Final Weighted ILP Model for Gate Fusion

Decision variables:

$$x_j \in \{0,1\} \quad \text{select block } P_j.$$

Objective:

$$\min \sum_{j=1}^{m} w_j x_j.$$

Coverage constraints:

$$\sum_{j:g\in P_j} x_j = 1 \quad \forall g \in \mathcal{G}.$$

Acyclicity:

- Same cycle-elimination conditions as before.
- Enforces a DAG structure among fused blocks.

This is a principled model of gate fusion

Optimal partitions are now optimal in terms of **actual floating-point simulation cost**.

How to Evaluate the Approach

Implementation:

- Python implementation using a standard ILP modeling library.
- Solved with a modern commercial ILP solver under academic license.

Benchmarks:

- Standard quantum circuit suites used in prior work:
  - QFT, Ising / Heisenberg models, arithmetic circuits, etc.
- Circuits ranging from a few hundred to $\sim 70{,}000$ gates.

Baselines (for comparison):

- Kahn-style greedy topological partitioning.
- TDAG and GTQCP implementations.
- Qiskit's multi-qubit block collection.
- BQSKit (Quick variant).

Illustrative Example: Small Circuits

Setup: maximum qubit budget $k = 3$ (small, tightly constrained blocks).

Circuit	Gates	Kahn	GTQCP	ILP (ours)
qft_10	200	24	23	**21**
ising_model_13	633	38	30	**28**
heisenberg-16-20	1028	109	85	**84**

- All methods respect the same qubit bound.
- Heuristics get reasonably close, but:
  - ILP gives the provably optimal number of blocks.
- Even on these modest examples, the gap is visible.

Illustrative Example: Large Circuits

Example: large benchmark with $\sim 50{,}000$ gates.

Circuit	Gates	Kahn	GTQCP	ILP (ours)
urf5_280 ($k = 3$)	49,829	4953	4507	**4427**
urf5_280 ($k = 4$)	49,829	3986	3290	**3209**

- For large, realistic circuits, the ILP still improves on the best heuristics.
- Differences of a few percent may represent:
  - Thousands of fewer blocks,
  - Shorter depth or fewer communication boundaries.

Runtime: Is Exact Optimality Practical?

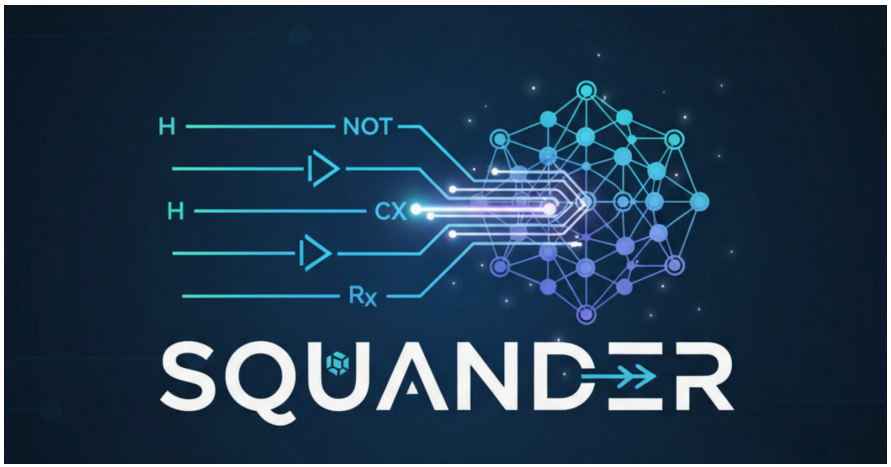
- We are solving a global optimization problem, so:
  - ILP is inevitably more expensive than a single greedy pass.
- Empirically:
  - Small circuits (hundreds of gates): solved almost instantly.
  - Medium circuits (thousands of gates): typically seconds.
  - Large circuits (up to $\sim 50\,000$ gates): minutes.
- Modern ILP solvers exploit:
  - Strong presolve,
  - Good cutting planes for set-cover–like formulations,
  - Parallel branch-and-bound.

Take-away

Exact optimality is not free, but it is *feasible* at scales that are already interesting to simulators and compilers.

How to Use This in Practice

- ILP-based partitioning is integrated into an open-source library for Sequential QUANtum gate DecomposER (SQUANDER).



- In practice:
  - For small and medium circuits, you can run the ILP directly.
  - For very large workloads, ILP can serve as:
    - A **benchmark tool** to evaluate heuristics.
    - A way to calibrate and improve heuristic cost models.
- Access to an industrial-strength solver (e.g., academic Gurobi license) is helpful for the largest instances.

Takeaway: the ground truth is now obtainable.

From Fusion to Decomposition

So far:

- We used ILP and weights (FLOPs + I/O) to find **optimal fusion blocks** for simulation.

Now we change perspective:

- Instead of minimizing floating-point cost, we want to:
  - **Decompose** a target unitary into CNOT + single-qubit gates,
  - on **all-to-all** or constrained (graph) topologies.
- The natural "weight" is now:

$$w(P_j) = \text{CNOT count in block } P_j$$

  instead of FLOPs.

Philosophy

Same ILP machinery, different cost model:

- Fusion: weights = simulation cost.
- Decomposition: weights = **CNOT counts** (or depth).

Why Pre-Decomposition Is Necessary

Key Idea: Partitioning operates on CNOTs + 1-qubit gates. To reason about communication, entanglement, and partition boundaries, we must express every multi-qubit gate in this basis.

Reasons this is essential:

- **Partition weights depend on CNOT counts.**
  - The ILP's objective uses the minimal number of CNOTs needed to implement a block.
  - Therefore, every gate must first be expanded into its known optimal CNOT form.
- **Partition boundaries must be between elementary operations.**
  - Multi-qubit gates spanning different partitions are ambiguous.
  - Once expanded to CNOTs, the bipartite structure becomes explicit.

Why Pre-Decomposition Is Necessary (cntd.)

- **All well-known 2- and 3-qubit gates have optimal decompositions.**
  - Library gates (e.g., CY, CU, CCX) have known best CNOT counts.
  - Using these avoids underestimating or overestimating partition cost.
- **Preprocessing drastically reduces ILP complexity.**
  - Without decomposition, the ILP must explore *all* equivalent realizations—exponential blowup.
  - With decomposition, only one canonical form enters the search.

Note: For full completeness, multiple possible decompositions may be tried, but this is a secondary refinement step.

Pre-Partitioning: Basic Decomposition of Standard Gates

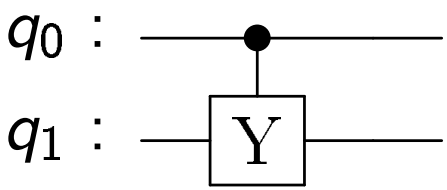
$$U(\theta,\phi,\lambda) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\phi}\sin\left(\frac{\theta}{2}\right) \\ e^{i\lambda}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{bmatrix} \quad q: \; \boxed{\mathrm{U}\left(\theta,\phi,\lambda\right)}$$

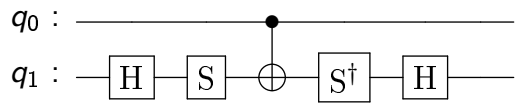
$$= \begin{bmatrix} e^{-\frac{i\phi}{2}} & 0 \\ 0 & e^{\frac{i\phi}{2}} \end{bmatrix} \cdot \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \cdot \begin{bmatrix} e^{-\frac{i\lambda}{2}} & 0 \\ 0 & e^{\frac{i\lambda}{2}} \end{bmatrix} \cdot \begin{bmatrix} e^{i\left(\frac{\lambda}{2}+\frac{\phi}{2}\right)} & 0 \\ 0 & e^{i\left(\frac{\lambda}{2}+\frac{\phi}{2}\right)} \end{bmatrix}$$

$$q: \; \boxed{\mathrm{R_Z}\left(\phi\right)} \; \boxed{\mathrm{R_Y}\left(\theta\right)} \; \boxed{\mathrm{R_Z}\left(\lambda\right)} \; \boxed{\mathrm{GP}\left(\frac{\phi+\lambda}{2}\right)}$$

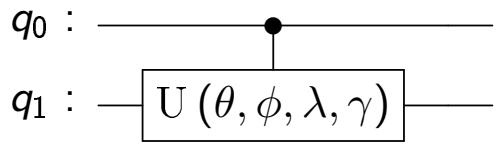
Pre-Partitioning: Decomposition of 2-qubit Standard Gates

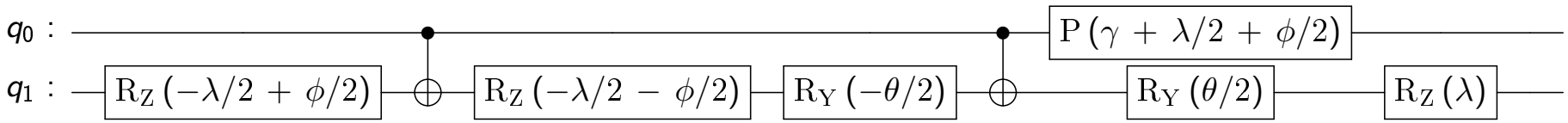
$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & 1 & 0 \\ 0 & i & 0 & 0 \end{bmatrix}$$





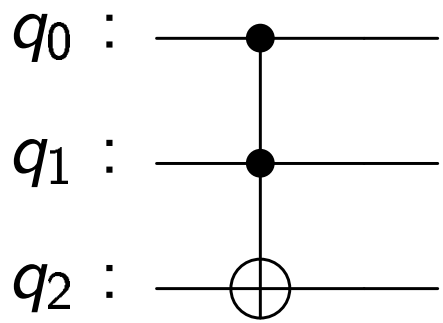
$$CU(\theta,\phi,\lambda,\gamma) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\gamma}\cos\left(\frac{\theta}{2}\right) & 0 & -e^{i(\gamma+\phi)}\sin\left(\frac{\theta}{2}\right) \\ 0 & 0 & 1 & 0 \\ 0 & e^{i(\gamma+\lambda)}\sin\left(\frac{\theta}{2}\right) & 0 & e^{i(\gamma+\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

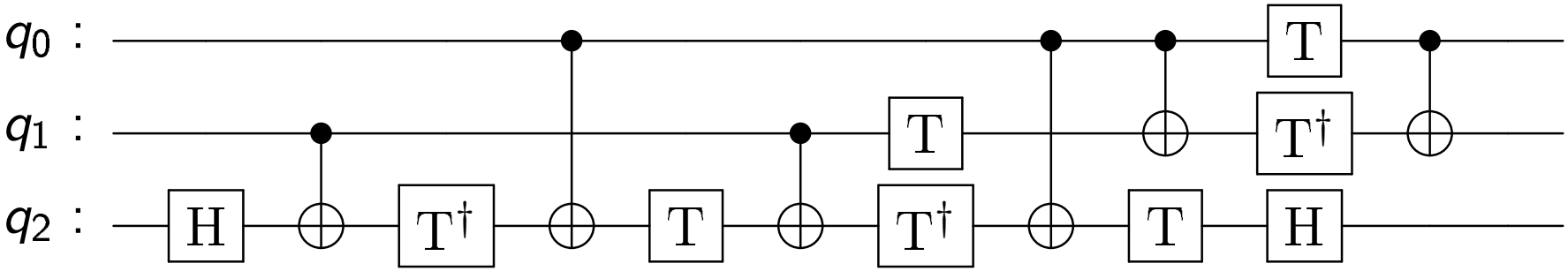




Pre-Partitioning: Decomposition of 3-qubit Standard Gates

$$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$





Linear Reversible Circuits and GL($n, 2$)

CNOT-only circuits on n qubits:

- Any sequence of CNOTs (over all qubits) implements a linear reversible map:

$$x \mapsto Ax \pmod 2, \quad A \in \mathrm{GL}(n, 2).$$

- Each CNOT corresponds to an elementary row operation over $\mathbb{F}_2$.
- The group of all such maps is the general linear group $\mathrm{GL}(n, 2)$.

OEIS A002884:

$$|\mathrm{GL}(n, 2)| = 1, 1, 6, 168, 20160, 9999360, \ldots$$

- Number of nonsingular $n \times n$ matrices over $\mathbb{F}_2$.
- Also: number of distinct linear Boolean / CNOT-only operations on n bits/qubits.

Interpretation

This sequence is the **search space** size for CNOT-only decompositions on n qubits.

OEIS A002884: Size of $\mathrm{GL}(n,2)$ and CNOT-Only Circuits

Definition (OEIS A002884):

- Number of reversible linear quantum operations on n qubits using only CNOT gates.

Closed form:

$$|\mathrm{GL}(n,2)| = \prod_{k=0}^{n-1} \left(2^n - 2^k\right)$$

Asymptotics: $|\mathrm{GL}(n,2)| \approx 2^{n^2}$ (up to lower-order factors).

First values (OEIS A002884):

n	$\lvert\mathrm{GL}(n,2)\rvert$	$\log_{10}$
1	1	0.0
2	6	0.78
3	168	2.23
4	20160	4.30
5	9 999 360	6.00
6	20 158 709 760	10.30
7	1.64×10^{14}	14.2
8	5.35×10^{18}	18.7

Interpretation for CNOT synthesis

Each element of $\mathrm{GL}(n,2)$ corresponds to a distinct CNOT-only linear reversible circuit on n qubits.

Key message

The search space for CNOT-only decompositions grows *super fast* with n: exhaustive search is hopeless beyond very small n.

OEIS A002884 and Depth-Ordered CNOT Enumeration

Radcliffe's interpretation (OEIS comment):

- A002884 also counts
  "The number of Boolean operations on n bits, or quantum operations on n qubits, that can be constructed using only CNOT gates."

Why this matters for us:

- $\mathrm{GL}(n, 2)$ is the *full* CNOT-only space.
- Our BFS enumeration over CNOTs:
  - explores $\mathrm{GL}(n, 2)$ in increasing depth,
  - assigns a minimal CNOT count to each linear map.
- These minimal depths become: weights $w(A) =$ CNOT count, used in our weighted ILP for decomposition.

Conceptual picture:

- Depth 0: identity only.
- Depth 1: all single-CNOT maps.
- Depth 2: all maps reachable by at most 2 CNOTs, etc.

Philosophy

OEIS A002884 tells us how huge $\mathrm{GL}(n, 2)$ is. BFS-by-depth gives us:

- structure (minimal depth),
- a principled **cost metric** for CNOT-based blocks.

BFS Enumeration of CNOT Networks by Depth

Idea: Enumerate elements of $\mathrm{GL}(n, 2)$ in order of CNOT depth.

- Represent each linear reversible map by an $n \times n$ binary matrix A.
- Start from the identity I at depth 0.
- At each BFS level:
  - Apply all allowed CNOTs (according to the topology) to each matrix.
  - Record each *new* matrix in a visited set.
  - The BFS level gives the **minimal CNOT count** for that map.

Weight for decomposition

For a block implementing a linear map A:

$$w(A) = \text{minimal CNOT count from BFS depth.}$$

These CNOT counts replace FLOP-based weights in the ILP objective.

BFS Level Expansion (Topology-Aware)

Algorithm 1: BFS_Expand_Level(frontier, visited, topology)

Input : frontier: set of matrices $A \in \mathrm{GL}(n, 2)$ at current depth
visited: set of matrices already discovered
topology: set of allowed CNOT edges (i, j)
Output: next_frontier: matrices at the next CNOT depth

```
next_frontier ← ∅;
foreach A ∈ frontier do
    foreach (i, j) ∈ topology do
        foreach (c, t) ∈ {(i, j), (j, i)} do
            B ← apply_CNOT(A, control = c, target = t);
            if B ∉ visited then
                visited ← visited ∪ {B};
                record parent / sequence information for B;
                next_frontier ← next_frontier ∪ {B};
return next_frontier;
```

Key properties:

- BFS guarantees minimal CNOT depth for each reachable linear map.
- topology encodes which CNOTs are allowed (line, grid, all-to-all, arbitrary graph).
- Applicable to any reversible linear map over $\mathrm{GL}(n, 2)$.

Symmetry Reduction for Topology-Aware Enumeration

Context: For the non-$\mathrm{GL}(n, 2)$ variant, we enumerate sequences of CNOT edges rather than binary matrices. To avoid redundant sequences, we enforce:

1. Limit on repeated CNOTs

- For any edge $p \in$ topology, we allow at most three consecutive occurrences of p in a sequence.
- If the last three steps in a prefix are all equal to p, we do not extend the prefix by p again.
- This cuts away long, uninteresting chains of identical CNOTs and reduces the search space.

2. Canonical topological order (prefix test)

We define a partial order between CNOTs by qubit usage: gate k depends on gate p if they share a qubit and p appears earlier.

Symmetry Reduction Canonical Prefix Checking

Algorithm 2: IsCanonicalPrefix($e_0, \ldots, e_{m-1}$)

Input : sequence of CNOT edges $e_k = \{i_k, j_k\}$
Output: true if prefix is canonical, false otherwise
Build a dependency DAG over positions $0, \ldots, m-1$:
for each gate k and each qubit $q \in e_k$, add an edge $p \to k$ from the last gate p that used q.
Compute in-degrees and initialize a priority queue Q with all nodes of in-degree 0, ordered by a fixed lexicographic order on edges e_k.
for $pos = 0$ **to** $m-1$ **do**
    If Q is empty: return false (malformed DAG).
    Extract from Q the gate index u with lexicographically minimal edge e_u.
    if $u \neq pos$ **then**
        return false ; `// sequence deviates from canonical topological order`
    For each successor v of u: decrement in-degree; when it reaches 0, insert v into Q.
Return true.

Usage: A candidate sequence is only kept if *every prefix* is canonical; otherwise, it is rejected as a symmetric duplicate.

From CNOT Sequences to a U3+CNOT Fabric

Input: A sequence of unordered CNOT pairs

$$\text{pairs} = \left[(i_0, j_0), (i_1, j_1), \ldots, (i_{L-1}, j_{L-1})\right]$$

over n qubits.

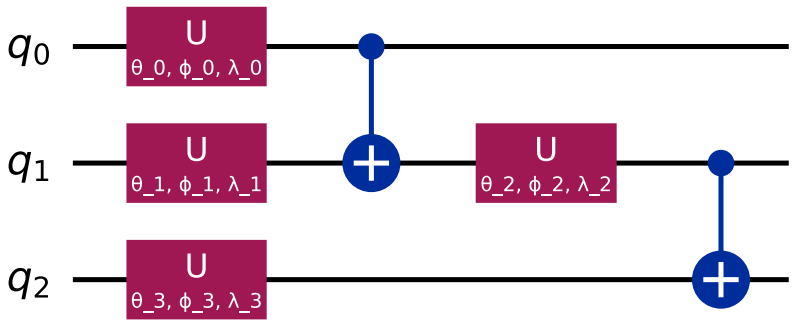
Construction of the fabric:

- For each pair (i_k, j_k) in order:
  - apply a parameterized $U3$ gate on qubit i_k,
  - apply a parameterized $U3$ gate on qubit j_k,
  - apply a CNOT with control i_k and target j_k.
- Optionally, add a finalizing layer of single-qubit $U3$ gates on all n qubits at the end.

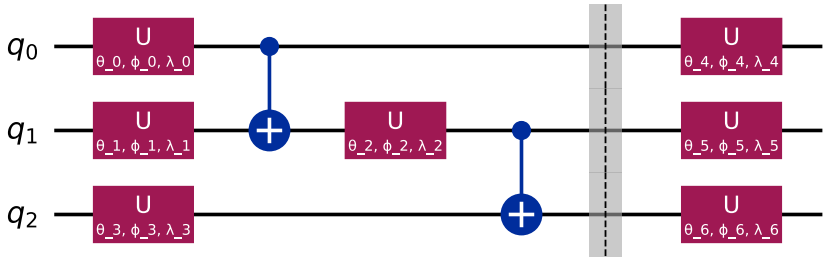
This yields a parametric $\mathrm{U3} + \mathrm{CNOT}$ ansatz block associated to each topology-aware CNOT sequence discovered by BFS.

Example U3+CNOT Fabric

Without finalizing layer



With finalizing U3 layer



Weighted ILP for CNOT-Based Decomposition

For a fixed topology and target unitary:

- We build candidate blocks P_j that combine CNOT layers and single-qubit gates.
- Each block P_j is assigned a CNOT-based weight w_j from BFS depth.

ILP objective (decomposition version):

$$\min \sum_{j=1}^{m} w_j x_j$$

Subject to:

- Each gate is covered exactly once (as before).
- Partition interaction graph is acyclic (as before).
- Qubit budget / topology constraints are respected.

Interpretation

We now get globally optimal partitions *with respect to CNOT cost* instead of simulation cost.

Motivation: Measuring Entanglement of a Block

For decomposition we also care about:

- **How entangling** a block is across each bipartition.
- For gradient-based optimization, we need a smooth cost function that reflects entanglement structure.

Tool: Operator Schmidt Rank (OSR)

- Given U acting on n qubits and a cut $A|B$:

$$U = \sum_k \lambda_k A_k \otimes B_k,$$

  where λ_k are singular values of a reshaped matrix.
- The number of significant λ_k quantifies entanglement of U across the cut $A|B$.

Constructing the OSR Matrix M for a Cut

Let U be a $2^n \times 2^n$ unitary in row-major order, and let $A \subset \{0, \ldots, n-1\}$ be the qubits on one side of the cut.

Dimensions:

$$d_A = 2^{|A|}, \quad d_B = 2^{n-|A|},$$

and we build:

$$M \in \mathbb{C}^{(d_A^2)\times(d_B^2)}.$$

Index mapping:

$$M_{(a' d_A + a),\,(b' d_B + b)} = U_{(a',b'),(a,b)},$$

where:

- (a, b) encodes input basis indices on A and B,
- (a', b') encodes output basis indices on A and B.

OSR singular values:

$$M = U_{\text{osr}}\, \Sigma\, V_{\text{osr}}^{\dagger}, \quad \Sigma = \operatorname{diag}(S_0, S_1, \ldots).$$

After normalization, the S_i capture the operator Schmidt spectrum.

Constructing the OSR Matrix: Algorithmic View

Algorithm 3: Build M for a bipartition $A|B$

Input : Unitarty $U \in \mathbb{C}^{2^n \times 2^n}$, cut $A \subset \{0, \ldots, n-1\}$
Output: OSR matrix $M \in \mathbb{C}^{(d_A^2) \times (d_B^2)}$
Compute complementary set B and dimensions $d_A = 2^{|A|}$, $d_B = 2^{|B|}$;
Initialize M as a $(d_A^2) \times (d_B^2)$ complex matrix;
foreach *basis pair* (a, b) *on* A *and* B **do**
    foreach *basis pair* (a', b') *on* A *and* B **do**
        Set row index $r = a' d_A + a$;
        Set column index $c = b' d_B + b$;
        Set $M_{r,c} = U_{(a',b'),(a,b)}$;

OSR Across All Cuts

For an n-qubit block, we consider **all nontrivial bipartitions**:

$$A|B, \quad A \subset \{0, \ldots, n-1\}, \; 1 \leq |A| \leq \left\lfloor \frac{n}{2} \right\rfloor,$$

modulo complement symmetry.

For each cut c:

1. Build M_c via the OSR reshaping.
2. Compute singular values $S^{(c)}$ of M_c (SVD).
3. Normalize: $S_i^{(c)} \leftarrow S_i^{(c)} / \|U\|_F$.

The OSR spectrum $\{S^{(c)}\}$ provides a rich view of the entanglement structure of the block across all bipartitions.

Philosophy of the OSR Cost

Desired properties:

- **Differentiable**: usable in gradient-based optimization.
- **Sensitive to entangling tails**: small but nonzero singular values.
- **Scale-aware**: focus on "dyadic" ranks $1, 2, 4, \ldots$ (powers of two).

Heuristic design choices:

- Work with the normalized singular values $S = (S_0, S_1, \ldots)$.
- Emphasize entries at indices $1, 2, 4, \ldots$ (dyadic positions).
- Use a decaying weight ρ^k to discount higher-order dyadics.
- Subtract a small tolerance multiple of S_0 to be robust to numerical noise.

Goal

Penalize "residual" entanglement beyond the leading Schmidt components across all cuts.

Per-Cut Tail Loss

For a single cut with singular values $S = (S_0, S_1, \dots)$:

$$\text{tail\_loss}(S) = \sum_{k=0}^{K-1} \rho^{K-1-k} \left(S_{2^k} - S_0 \cdot \text{tol} \right)^2$$

where:

- $K = \lceil \log_2 |S| \rceil$ is the maximum dyadic scale,
- $\rho \in (0, 1)$ is a decay factor (e.g. $\rho = 0.1$),
- `tol` is a small tolerance to ignore numerical noise.

Interpretation:

- Large values at dyadic positions 2^k indicate higher effective OSR rank.
- The cost encourages singular spectra that decay quickly (low entangling power).

Aggregating Over Cuts: Softmax

For all cuts $c = 1, \ldots, C$, with per-cut tail losses L_c:

Softmax aggregation:

$$L_{\mathrm{OSR}} = \tau \log \sum_{c=1}^{C} \exp\left(\frac{L_c - m}{\tau}\right) + m,$$

where:

- $m = \max_c L_c$,
- $\tau > 0$ is a "temperature" parameter (e.g. 10^{-2}).

Properties:

- As $\tau \to 0$, $L_{\mathrm{OSR}} \approx \max_c L_c$ (worst-cut behavior).
- For larger τ, we get a smoother, more averaged penalty.

This L_{OSR} becomes our **entanglement cost** for the block.

Gradient w.r.t. Singular Values

For a single cut:

$$L_c(S) = \sum_{k=0}^{K-1} \rho^{K-1-k} \Big(S_{2^k} - S_0 \cdot \mathrm{tol} \Big)^2 .$$

The derivative at dyadic positions (schematically):

$$\frac{\partial L_c}{\partial S_{2^k}} = 2\, \rho^{K-1-k} \Big(S_{2^k} - S_0 \cdot \mathrm{tol} \Big).$$

Non-dyadic indices receive zero gradient from this cost.

For the softmax aggregation:

$$\frac{\partial L_{\mathrm{OSR}}}{\partial L_c} = w_c = \frac{\exp((L_c - m)/\tau)}{\sum_{c'} \exp((L_{c'} - m)/\tau)},$$

so:

$$\frac{\partial L_{\mathrm{OSR}}}{\partial S^{(c)}} = w_c \cdot \frac{\partial L_c}{\partial S^{(c)}}.$$

Backpropagating Through SVD (Conceptual)

For each cut c:

$$M_c = U_c \, \Sigma_c \, V_c^\dagger,$$

with singular values $S^{(c)}$ on the diagonal of Σ_c.

Given $\frac{\partial L}{\partial S^{(c)}}$:

- We can build the gradient w.r.t. M_c using standard SVD derivative formulas.
- Then we map the gradient on M_c back to U via the OSR reshaping (inverse of the construction of M_c).

End result

We obtain $\frac{\partial L_{\text{OSR}}}{\partial U}$ as a full $2^n \times 2^n$ matrix, which can be paired with:

$$\frac{\partial U}{\partial \theta_i}$$

for each circuit parameter θ_i to obtain parameter gradients.

Gradient Accumulation for OSR: High-Level Algorithm

Algorithm 4: Compute $\frac{\partial L_{\text{OSR}}}{\partial U}$

Input : U: an $2^n \times 2^n$ unitary matrix
$\mathcal{C}$: list of bipartition cuts

Output: $\frac{\partial L}{\partial U}$: gradient of OSR loss w.r.t. U

1. **Construct OSR matrices. foreach** *cut* $c \in \mathcal{C}$ **do**
   - Form matrix M_c by reshaping U according to cut c;
   - Compute SVD: $M_c = U_c \Sigma_c V_c^\dagger$;
   - Let $S^{(c)}$ be the diagonal singular-value vector of Σ_c;

2. **Compute gradients w.r.t. singular values.** Apply the chosen OSR loss (e.g. softmax tail) to each $S^{(c)}$ to obtain $\frac{\partial L}{\partial S^{(c)}}$;

3. **Backpropagate through each SVD. foreach** *cut* $c \in \mathcal{C}$ **do**
   - Use SVD derivative identities to compute $\frac{\partial L}{\partial M_c}$ from $\frac{\partial L}{\partial S^{(c)}}$;
   - Map $\frac{\partial L}{\partial M_c}$ back to $\frac{\partial L}{\partial U}$ via the inverse reshape of cut c;

4. **Aggregate contributions.** Sum the per-cut gradients over all cuts:

$$\frac{\partial L}{\partial U} = \sum_{c \in \mathcal{C}} \text{reshape}_c^{-1}\left(\frac{\partial L}{\partial M_c}\right).$$

Putting It All Together

1. **Partition** the circuit with ILP:
   - Fusion: weights = FLOPs + I/O.
   - Decomposition: weights = CNOT counts (from GL$(n, 2)$ BFS).
2. **Within each block**, optimize a parameterized circuit:
   - Use OSR-based entanglement cost across all cuts.
   - Use SVD + gradient backprop to get $\nabla_\theta L_{\text{OSR}}$.
3. **End result:**
   - A decomposition that is:
     - Topology-aware,
     - CNOT-efficient,
     - And controlled by a mathematically well-founded entanglement cost.

Next: examples and hands on for partitioning, fusion, simulation, GL(n, 2), OSR, decomposition.