# Fast Quantum Computation Optimisation using Concurrent Graph Rewriting

Luca Mondada
University of Oxford
Oxford, UK
Quantinuum Ltd
Cambridge, UK
luca.mondada@cs.ox.ac.uk

Agustín Borgna
Quantinuum Ltd
Cambridge, UK
agustin.borgna@quantinuum.com

## Overview

**Graph rewriting** is among the simplest and most general techniques for quantum circuit and quantum computation optimisation. A graph rewriting system in the context of quantum circuits is described by a set of rewrite rules, given as pairs of pattern and replacement circuits—respectively the left and right hand side of the rule. Optimisation of a circuit is performed by repeatedly finding subcircuits of the input that match a rewrite rule left hand side and replacing them with the corresponding right hand side.

Complete equational theories for quantum circuits [2, 1] give us sets of rewrite rules that can be used for rewriting, with theoretical guarantees that any two equivalent circuits are connected to one another by a finite sequence of rewrites. However, the construction of [2] will in general require a number of rewrites exponential in the size of the input circuit, rendering it an unsuitable approach for optimisation.

**Large search space.** Overcomplete sets of rewrite rules alleviate this issue [13, 12] by introducing additional rewrite rules, at the cost of widening the search space. In the absence of good search heuristics, this results in slow optimisation progress. As input sizes grow, the number of pattern matches—and thus the potential rewrites—increases, further compounding the challenges of optimisation within the large search space. The inherently sequential nature of such backtracking search algorithms also make them hard to parallelise [15].

Table 1 illustrates these challenges on an example by measuring the branching factor of the search space, i.e. the number of candidate rewrites that can be performed as an optimisation step[1]. On a circuit with 518 gates, each optimisation step must consider 473.6 options on average. This increases linearly with circuit size.

The table also shows a mitigation strategy: for searches over local cost functions such as `CX` or `T` gate count, the problem can be divided into subproblems by optimising subcircuits of the input and combining the results. This is very effective at reducing the branching factor and can be processed in parallel. However, it also results in a 5% performance decrease as circuit splitting introduces boundaries that reduce the set of rewrites considered during optimisation.

| | `CX` cost reduction | Average Branching Factor | Nb of cores |
|---|---|---|---|
| *No circuit splitting* | 32% | 473.6 | 1 |
| *Circuit splitting (10 splits)* | 27% | 38.4 | 10 |

Table 1: Comparing circuit optimisation performance using rewrites, with and without circuit splitting. The optimisation was run on the `barenco_tof_10` circuit (19 qubits, 518 gates) using the rewrite rules obtained from the equivalence classes of circuits with up to 3 qubits and 6 gates on the Nam gateset, as generated by TASO (see [6] for details). Best circuit after 1000 circuits were explored (per thread limit).

[1] These results can be reproduced using the `badger-optimiser` available at `https://github.com/CQCL/tket2`.

Submitted to:
IWQC Workshop 2024

**A novel data structure.** We propose concurrent graph rewriting as a solution that offers the parallelism and lower branching factors of circuit splitting without trading off optimisation performance. Our data structure stores rewrites that can be applied either on the input circuit directly or following a sequence of other rewrites. This data is encoded in such a way that any output circuit that can be obtained by a sequence of rewrites on the input can be extracted efficiently.

As new rewrite candidates are explored, it is sufficient to consider transformations that overlap with previous rewrites, in effect reducing the branching factor of the search space in a very similar way to circuit splitting. However, the "splits" of the circuit are dynamic and are modified as required to ensure that all possible rewrites are considered, thus never encurring a performance penalty.

**Comparison with equality saturation.** This idea relates to the equality saturation technique as used in term rewriting [9, 11]. Equality saturation leverages the hierarchical structure encoded by the algebraic syntax tree (AST) of a term to store the sets of equivalent terms at the root of every subterm in the AST. Equivalent subterms can thus repeatedly be added to the data structure until "saturation", when all reachable terms have been discovered. For local cost functions, the "extraction" step that follows and yields the final best term is a simple greedy traversal of the saturated data structure.

By contrast, the lack of hierarchy in directed acyclic graphs—and graphs more generally—makes it impossible to map every subgraph injectively to root nodes. As a result, concurrent graph rewriting offers weakened theoretical guarantees, and in particular is not guaranteed to saturate, even for small rewriting systems.

It preserves however two of the main qualities of equality saturation relevant in practice. It provides on the one hand an efficient representation of the rewriting space and transforms an optimisation problem over a branching space of rewrites into an extraction problem that can be solved with an SMT solver. It further makes multi-threaded exploration of the rewriting space easy as the end state of the data structure is independent of the rewriting order (data is only added, never removed). Exploration could be done in distributed systems by maintaining local copies of the data structure in each thread, with a synchronisation protocol as implemented in conflict-free replicated data types (CRDTs) [8, 7].

## Detailed Description

We suppose in this section that the quantum computation is expressed as a port graph, that is a multigraph in which each end of an edge is labelled by a port. Vertices themselves have labels that indicate their gate type. For simplicity we assume that every incident edge to a vertex has a unique port, although this restriction is not required by the data structure.

**Data Structure.** Our concurrent graph rewriting data structure $\mathscr{D}$ stores a single source directed acyclic graph of *local graph diffs*—in the following just diffs. Each diff $D = (R, B, \varphi)$ is given by a replacement port graph $R$, a list of boundary ports $B$ along with a map $\varphi : B \to \cup_{P \in parents}\ ports(P)$ from the boundary ports to ports in the parent diffs. Boundary ports are additional ports on the vertices $V(R)$ that are not in $R$—they can be thought of as "open" ports, i.e. ports not connected to edges. The $ports(D)$ of a diff $D$ is then the union of the ports of $R$ and its boundary ports $B$.

Edges between diffs store the subgraph $H$ of the parent replacement graph the rewritte applies to. The boundary ports of $H$ are defined as the boundary ports of $G$ on vertices in $V(H)$, as well as ports on $V(H)$ that are incident to edges in $E(G) \setminus E(H)$. The image of the boundary map $\varphi$ of the child must be a subset of the boundary ports of the subgraphs $H$ in the parent diffs. Figure 1 summarises the data structure on an example.

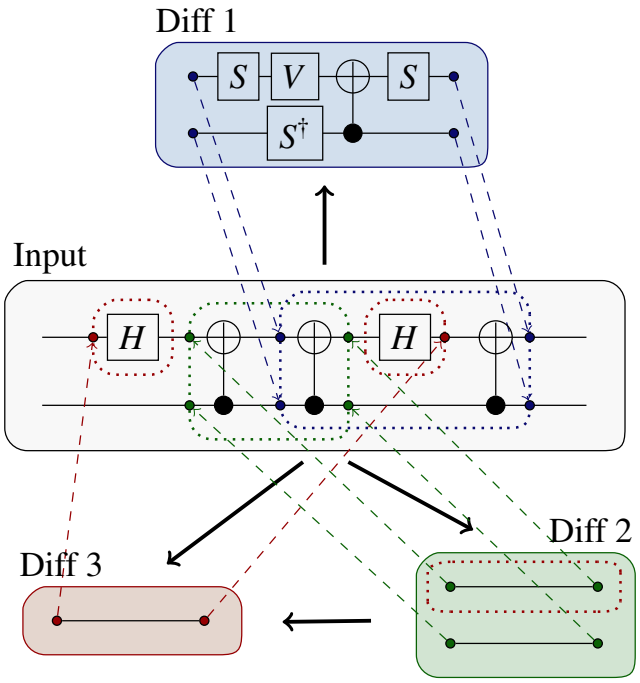


Figure 1: Example of the data structure $\mathscr{D}$ after three rewrites on an input circuit (grey). Circuits within the input and diffs are the replacement graphs $R$. The boundary ports $B$ and their image under the boundary map $\varphi$ are given by the small coloured circles. The boundary map $\varphi$ itself is indicated by the dashed edges. The thick black edges indicate the diff rewrite history; the dotted subgraphs on the diffs are the edges' respective subgraphs $H$. Note that diff 3 rewrites subgraphs in both the input and in diff 2 (and has hence two incoming edges).

For an input port graph $G$, the data structure $\mathscr{D}$ is initialised with a single diff with replacement graph $G$ and the empty boundary $B = \varnothing$. A rewrite can then be applied to a set of replacement graphs from previous diffs $G_1, \ldots, G_k$ (subject to some constraints, see below) by adding it to $\mathscr{D}$ as a child diff of the diffs of $G_1, \ldots, G_k$. The new diff's replacement graph and boundary are given by the right hand side of the applied rewrite rule. The subgraphs $H_1, \ldots, H_k$ on the edges from the parent diffs, as well as the boundary map $\varphi$ from $B$ to their boundary ports, are given by the match of the rewrite left hand side in the parent replacement graphs.

Finally note that for any diff $D$ we can define the map $\varphi^*$ from the boundary ports $B$ of $D$ to non-boundary ports in ancestors of $D$: the map $\varphi$ of $D$ will map a boundary port in $B$ to a port in one of its parent diffs, which is either a non-boundary port—we are done—, or a boundary port—in which case we can apply the parent's boundary map $\varphi$ and proceed recursively until a non-boundary port is reached.

**Supported operations.** The rewriting operations are enabled by two operations on the data structure. FINDALLOPPOSITES allows to pattern match graphs across multiple diffs, by finding diffs that include opposite ends of an edge in their boundaries. The other is a compatibility predicate that verifies that the rewrites do not overlap and can thus be applied in parallel.

**FINDALLOPPOSITES$(D, p)$** For a diff $D \in \mathscr{D}$ and a boundary port $p \in B_D$ of $D$, return all pairs $(D', p')$ such that $D' \in \mathscr{D}$, $p' \in B_{D'}$ is a boundary port of $D'$ and $\varphi^*(p)$ is connected to $\varphi^*(p')$ by an edge in the replacement graph of a common ancestor of $D$ and $D'$.

**ARECOMPATIBLE($\{D_1, D_2, \ldots, D_k\}$)** A predicate that returns true iff for the subgraph $A \subseteq \mathscr{D}$ that contains all the diffs in $\mathscr{D}$ that are ancestors of $D_i$ for some $1 \leq i \leq k$, all rewrites are compatible, that is, for every diff $D \in A$, the subgraphs on the outgoing edges of $D$ that are in $A$ are vertex-disjoint.

**Pattern matching and rewriting.** Using the above operations, we can proceed with matching all the left hand sides in our rewriting system across all diffs in $\mathscr{D}$ (assuming patterns are connected graphs): starting at some vertex in some diff $D \in \mathscr{D}$, we can match one pattern edge at a time, either matching it to an edge in the replacement graph of $D$ or to two opposite ports $p$ and $p'$, with $p \in B_D$ in the boundary of $D$ and $p'$ in the boundary of some other diff $D'$ as returned by FINDALLOPPOSITES$(D, p)$. Once we have found a map for every edge of a rewrite rule left hand side, we must check that all diffs in the match are compatible using the ARECOMPATIBLE predicate, and if so, we create a new diff with the right hand side of the rewrite rule as replacement graph, mapping the boundary of the rewrite rule right hand side to the boundary of the matched subgraph.

Every call to FINDALLOPPOSITES can be performed in constant time by precomputing $\varphi^*$ and its inverse (note $\varphi^*$ is not injective, thus the inverse is a relation that map ports in $D$ to sets of boundary ports in descendants of $D$). The ARECOMPATIBLE predicate is evaluated by iterating over $i = 1 \ldots k$ and inductively checking that $D_i$ is compatible with the set of diffs $\{D_1, \ldots, D_{i-1}\}$. This can be done by computing the least common ancestors of $D_i$ with the subgraph of ancestors of $\{D_1, \ldots, D_{i-1}\}$ and checking that the outgoing edges are compatible. In the worst case, this will take time proportional to the total number of diffs $|\mathscr{D}|$. This can be mitigated in practice by squashing old rewrites into fewer larger diffs, thus reducing $|\mathscr{D}|$ whenever surpassing a threshold. Caching computed compatibility relations between diffs might also result in significant speedups.

**Extraction.** Once the rewriting space has been explored, the final output that minimises a given (local) cost function must be extracted. A set of diffs in $\mathscr{D}$ can be extracted into a single output graph precisely when the diffs in the set are compatible: the output is obtained by starting from the root replacement graph, and applying every rewrite in the subgraph of $\mathscr{D}$ formed by the ancestors of the diff set, in topological order. Compatibility ensures that no two rewrites to be applied on any one diff will overlap. Extraction therefore corresponds to finding the subset of compatible diffs in $\mathscr{D}$ that minimises the cost function when combined.

Assuming the cost function is of the form $f(G) = \sum_{v \in V(G)} c(v)$ for some cost $c(\cdot)$ for each vertex $v$ in $G$, we can easily express the problem as a constrained optimisation problem. We assign every diff $D \in \mathscr{D}$ a cost $w_D$, given by the cost delta of the rewrite rule $f(R) - f(L)$, where $L$ and $R$ are the left and right hand side of the rewrite rule. We then introduce a boolean variable $x_D$ for each diff $D \in \mathscr{D}$ and interpret $x_D = 1$ as applying the rewrite $D$. Every edge from $P$ to $D$ in $\mathscr{D}$ is encoded as a clause $x_D \Rightarrow x_P$, or equivalently $\neg x_P \vee x_D$, guaranteeing that if $D$ is applied, then all its ancestors must be applied as well. Finally, we add a clause $\neg(x_D \wedge x_{D'})$ for every pair of siblings $D$ and $D'$ that act on overlapping subgraphs. Any assignment of the variables $x_D$ that satisfies all the above clauses is a valid set of compatible diffs in $\widetilde{\mathscr{D}}$. To minimise the cost function we must minimise the total cost

$$\sum_{D \in \mathscr{D}} w_D x_D. \tag{1}$$

This can be easily solved using an SMT solver. For integer-valued cost functions, eq. (1) can be simplified further such that $w_D = -1$ for all $D$ by merging together diffs with cost larger than $-1$ and introducing dummy variables to break up diffs with costs smaller than $-1$. The resulting problem is a boolean satisfiability problem SAT, with the additional objective of maximising the number of active variables.

## Additional considerations

**Ensuring progress.** Whilst in theory the order in which rewrites are explored does not change the end state of the data structure, in practice the presence of a timeout on exploration time means that the most promising search directions should be prioritised and explored first.

In naive backtracking search, it is natural to introduce a progress timeout that terminates the optimisation early if no cost function reduction is obtained over the course of a timeout period. We suggest using a similar progress-based strategy to prioritise the most promising search directions. For each diff we can keep track of how much progress has been made over the last rewrites and use these values to order the diffs in a priority queue. Concretely, in the case of quantum circuits with `CX` or `T`-count optimisation, we keep track of the number of rewrites since the cost function was last decreased and use this value to order the diffs, prioritising diffs that have decreased the cost function most recently.

Another important optimisation in the search strategy is to prune the search whenever an already visited diff is found. This can be achieved by keeping track of the set of visited diffs and skipping any diff that is already in the set. Any hash function of diffs that would be used for this purpose must be independent of the rewrite history. A simple option would be to use the extraction procedure to obtain the rewritten port graph corresponding to the diff, which can in turn be hashed by a standard hashing function. This would be expensive however as every rewrite would trigger a hashing computation of a data structure of size of the order of the input $|G|$. This could be improved on by devising updateable hash functions that can compute the new hash value given the old hash value and the rewrite that is applied.

**Distributed graph rewriting.** Finally, we sketch how the exploration phase could be parallelised in a distributed environment. Assuming there is a fixed and global set of rewrite rules, we can assign a unique ID to each rewrite rule, to each vertex within the replacement graphs of the rewrite rules and to each boundary port. Using these IDs, an exploration node can describe a diff to other nodes by communicating its exact rewrite history (the DAG of rewrite rule IDs that were applied, along with the subgraph the rewrites were applied on). Upon receiving a diff description from another node, a compute node can check if the diff and its rewrite history are already in its local copy of $\mathscr{D}$, and insert it if it is not. To avoid large communication overheads, this protocol can be tuned to only broadcast diffs that are particularly promising or after a certain rewrite depth has been reached. Exploration should also be randomised in this case to avoid duplicating the same search over multiple compute nodes.

## Conclusion and Next Steps

In summary, this document proposes a novel data structure for concurrent graph rewriting. Changes on one subgraph can easily be propagated to distributed nodes and combined with concurrent changes to other parts of the graph. Furthermore, FINDALLOPPOSITES queries and the ARECOMPATIBLE predicate enable rewrites that span neighbouring subgraphs: as a result, any sequence of rewrites that is possible on a graph $G$ will be possible to perform on $\mathscr{D}$. However, only rewrites within a neighbourhood of a diff will be considered. The branching factor of the search space will thus be reduced, especially for large input graphs $G$. The combination of parallelisation and smaller search space should result in speedups.

The implementation of the ideas sketched in this document is currently under development. We expect to have first working examples in the next weeks and code ready to be shared in time for the workshop, along with early benchmarks.

## References


[1] Alexandre Clément, Noé Delorme & Simon Perdrix (2024): *Minimal Equational Theories for Quantum Circuits*. arXiv:2311.07476.

[2] Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix & Benoît Valiron (2023): *A Complete Equational Theory for Quantum Circuits*. In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, doi:10.1109/lics56636.2023.10175801.

[3] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: an efficient SMT solver*. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, p. 337–340.

[4] Peter E. Hart, Nils J. Nilsson & Bertram Raphael (1968): *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. *IEEE Transactions on Systems Science and Cybernetics* 4(2), pp. 100–107, doi:10.1109/TSSC.1968.300136.

[5] Gerard Huet (1977): *Conflunt reductions: Abstract properties and applications to term rewriting systems*. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 30–45, doi:10.1109/SFCS.1977.9.

[6] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia & Alex Aiken (2019): *TASO: optimizing deep learning computation with automatic generation of graph substitutions*. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, Association for Computing Machinery, New York, NY, USA, p. 47–62, doi:10.1145/3341301.3359630.

[7] Giovanna Márk Jelasity, Di Marzo Serugendo, Marie-Pierre Gleizes & Anthony Karageorgos (2011): *Gossip*, pp. 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-17348-6_7.

[8] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim & Joonwon Lee (2011): *Replicated abstract data types: Building blocks for collaborative applications*. *J. Parallel Distrib. Comput.* 71(3), p. 354–368, doi:10.1016/j.jpdc.2010.12.006.

[9] Ross Tate, Michael Stepp, Zachary Tatlock & Sorin Lerner (2009): *Equality saturation: a new approach to optimization*. *SIGPLAN Not.* 44(1), p. 264–276, doi:10.1145/1594834.1480915.

[10] Ross Tate, Michael Stepp, Zachary Tatlock & Sorin Lerner (2009): *Equality saturation: a new approach to optimization*. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, Association for Computing Machinery, New York, NY, USA, p. 264–276, doi:10.1145/1480881.1480915.

[11] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock & Pavel Panchekha (2021): *egg: Fast and extensible equality saturation*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1145/3434304.

[12] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu & Aws Albarghouthi (2023): *Synthesizing Quantum-Circuit Optimizers*. *Proceedings of the ACM on Programming Languages* 7(PLDI), p. 835–859, doi:10.1145/3591254.

[13] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar & Zhihao Jia (2022): *Quartz: Superoptimization of Quantum circuits*. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, Association for Computing Machinery, New York, NY, USA, p. 625–640, doi:10.1145/3519939.3523433.

[14] Yichen Yang, Mangpo Phitchaya Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy & Jacques Pienaar (2021): *Equality Saturation for Tensor Graph Superoptimization*. *CoRR* abs/2101.01332. arXiv:2101.01332.

[15] Yichao Zhou & Jianyang Zeng (2015): *Massively parallel A* search on a GPU*. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, AAAI Press, p. 1248–1254, doi:10.1609/aaai.v29i1.9367.