



Novel Methods for Classical Simulation of Quantum Circuits via ZX-Calculus

Matthew Sutcliffe

Wolfson College



University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2025

Dedicated to my late grandfather, Len.

Acknowledgements

I would firstly like to express my gratitude to Aleks for his invaluable supervision throughout the course of my DPhil. I appreciate all the hours he kindly spared to rack his brains with mine as we worked through research problems together in his office. I can only apologise for all the whiteboard marker ink we must have gone through. I thank him also for introducing me to ZX-calculus, which has since become such a prominent part of my life, I can scarcely remember what it was like to close my eyes and not see ZX-diagrams. Lastly, his very helpful software packages, including PyZX, QuiZX, and TikZiT, have been imperative to my work and surely saved me countless hours of tedious effort.

I would next like to thank all my friends and colleagues from, and adjacent to, the Quantum Group, including John, Tein, Razin, Lia, Boldi, Tuomas, and particularly Richie and Alex, for all the engaging discussions — from the fascinating to the silly — and for their ever-friendly company. I look forward to their future research output and hope to continue collaborating with them on exciting projects.

Likewise, I owe considerable gratitude to the welcoming community at Wolfson College and later Oriel College. Too numerous to name, their company has made Oxford a place I've been happy to call home. Charlotte, Olivia, Thomas, Paul, Lila, Zhenlong, Michael, Michail, and too many more to list, have been delightful friends — ever warm, ever considerate, and ever entertaining. The punting trips, the college balls, the dinners, the board games, the firewalk, all the rounds of pool, and the many all-night hangouts at the common room are experiences I'll cherish for years to come. I couldn't have asked for a better group of people with whom to share these past few years.

As for Tom and Rose and all my other friends outside of Oxford, I appreciate them

being a steady constant throughout the busy and hectic times among my studies. Despite the distance, our online gaming sessions are always a very welcome break after a hard day's work.

I would also like to thank my partner, Miko, for her support over these past few years and for listening to all my ramblings about my research and for sharing with me all the commentary of hers. I'm happy to have shared a DPhil journey with her, and I look forward to more adventures together.

Lastly, I am endlessly grateful to my parents and family for supporting and encouraging me through not just my DPhil but my entire education. I have them to thank for being where I am today, fulfilling my dream of pursuing exciting quantum research.

Abstract

With the limitations of today's '*NISQ era*' quantum hardware, classical simulation of quantum computations is an essential tool for understanding the quantum advantage and optimising quantum algorithms and systems. However, without the features unique to quantum computers, simulating such computations with classical hardware is (believed to be) a necessarily inefficient endeavour. For a general class of quantum circuits, any classical simulator requires a resource overhead which grows exponentially with one metric or another. Two notable and widely used techniques are tensor contraction and stabiliser decomposition. The former suffers from both space and time complexities that scale exponentially with the interconnectedness (or *treewidth*) of the circuit, while the latter is limited by a time complexity that scales exponentially with the number of '*non-Clifford*' gates in the circuit. Recent years have seen the graphical language of ZX-calculus applied to this problem, with a particular growing body of research utilising its benefits to discover more efficient stabiliser decompositions. This in turn reduces the growth rate of the exponential time complexity, thereby rendering ever larger and more complex quantum circuits classically simulable.

This thesis expands upon this literature by presenting various new techniques and approaches by which the ZX-calculus may be used to classically simulate quantum circuits with improved efficiencies. This work includes a new perspective on optimising stabiliser decomposition strategies, focusing on improved heuristics and decomposition patterns rather than on discovering wholly new decompositions. Also included is extensive use of GPU hardware to parallelise the workload, particularly used in conjunction with a parameterisation of the ZX-calculus which allows large sets of similar circuits to be reasoned upon as single entities. Lastly, this thesis features a hybrid method of classical simulation which leverages of the strengths of both tensor contraction and stabiliser decomposition approaches, optimising a balance between these techniques. Ultimately, the chapters ahead demonstrate how these new techniques can greatly increase the scope of classical simulation by reducing the runtime of this task by orders of magnitude and providing a foundation for further research.

Preface

This thesis is based on the research I conducted as a DPhil student in the Quantum Group of the Department of Computer Science at the University of Oxford, under the supervision of Dr Aleks Kissinger, from October 2021 to December 2024.

In particular, this work comprises five research projects, listed here in chronological order:

1. Matthew Sutcliffe and Aleks Kissinger, ‘Fast classical simulation of quantum circuits via parametric rewriting in the ZX-calculus’ (March 2024), *Preprint (to appear in the proceedings of the 22nd International Conference on Quantum Physics and Logic, Varna, Bulgaria, 14-18 July 2025)*. Available at: <https://arxiv.org/abs/2403.06777>. (Sutcliffe & Kissinger, 2024a)
2. Matthew Sutcliffe and Aleks Kissinger, ‘Procedurally Optimised ZX-Diagram Cutting for Efficient T-Decomposition in Classical Simulation’ (March 2024), *Proceedings of the 21st International Conference on Quantum Physics and Logic, Buenos Aires, Argentina, 15-19 July 2024, Electronic Proceedings in Theoretical Computer Science 406, p.63-78, doi:10.4204/eptcs.406.3*. Available at: <https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?QPL2024.3>. (Sutcliffe & Kissinger, 2024b)
3. Matthew Sutcliffe, ‘Smarter k-Partitioning of ZX-Diagrams for Improved Quantum Circuit Simulation’ (September 2024), *Preprint*. Available at: <https://arxiv.org/abs/2409.00828>. (Sutcliffe, 2024c)
4. Alexander Koziell-Pipe, Richie Yeung, Matthew Sutcliffe, ‘Towards Faster Quantum Circuit Simulation Using Graph Decompositions, GNNs and Reinforcement Learning’ (October 2024), *Proceedings of the 4th Workshop on Mathematical Rea-*

soning and AI at NeurIPS'24, Vancouver, Canada, 10-15 December 2024. Available at: <https://openreview.net/forum?id=54060pbCKY>. (Koziell-Pipe, Yeung, & Sutcliffe, 2024)

5. Wira Azmoon Ahmad and Matthew Sutcliffe, 'Dynamic T-decomposition for classical simulation of quantum circuits' (December 2024), *Preprint (to appear in the International Journal of Modern Physics C)*. Available at: <https://arxiv.org/abs/2412.17182>. (Ahmad & Sutcliffe, 2024)

Of the collaborative efforts among these projects, the extent of their inclusion in this thesis (except where stated otherwise) is focused on my own contribution. Specifically, papers 1 to 3 were predominantly my own independent research, conducted under the supervision and guidance of Dr Aleks Kissinger. The research presented in these three papers appear in chapters 3, 5, and 4, respectively. Meanwhile, paper 4 was a joint collaborative project with two peers. The contents of this paper (except in brief reference) are omitted from this thesis, though are expected to feature prominently in the upcoming DPhil theses of Alex Koziell-Pipe (Koziell-Pipe, 2025) and Richie Yeung. Lastly, paper 5, featuring in chapter 6, represents work developed in collaboration with Wira Ahmad, as an extension of their MSc thesis (Ahmad, 2024) which I proposed and supervised.

To the best of the author's knowledge, the contents of this thesis are accurate and up to date at the point of submission in February 2025.

Preliminaries

This thesis relies upon a number of terms which, in colloquial use, may have various similar and loosely defined meanings. It is essential for the clarity herein that explicit and precise definitions are provided for these terms. Similarly, this thesis introduces a handful of new terms, notations, and conventions which require definition and explanation.

Firstly, there is much reference made to the *image* of a function or parametric expression. In common parlance, this is often used interchangeably with its *codomain*. However, it is important for the understanding of the chapters ahead that this term is used precisely and accurately:

Definition 1. *The ‘image’ of a function $f : X \rightarrow Y$ is the set of all values in Y that f maps elements of X to. Formally, the image of f is:*

$$\text{Image}(f) = \{f(x) \mid x \in X\} \subseteq Y \quad (1)$$

Essentially, $\text{Image}(f)$ denotes the set of possible values which $f(x)$ may return, given accepted inputs from its domain, $x \in X$. For instance, given a function $f(x) = 2x + 3$ with $x \in \{0, 1\}$, it follows that $\text{Image}(f) = \{3, 5\}$.

Throughout this thesis, and particularly within chapter 3, there is a subtle but important distinction maintained between *concrete* and *abstract* parameters:

Definition 2. *Concrete parameters are variables that represent specific values drawn from a defined set. These parameters essentially act as placeholders for values which are as yet unspecified but nevertheless fixed.*

$p \in P$ defines a concrete parameter p whose value is fixed and belonging to the set P .

Definition 3. *Abstract parameters are variables, restricted to a defined set, which do not represent specific values but rather the variable itself. These remain symbolic and un-evaluated.*

$\mathfrak{p} \overset{\star}{\in} P$ defines an abstract parameter \mathfrak{p} whose image is known, $\text{Image}(\mathfrak{p}) = P$, but its value is not fixed to any particular element of P .

For instance, $a \in \{0, 1\}$ describes a specific, but currently unspecified, Boolean value (either $a = 0$ or $a = 1$), whereas $\mathfrak{a} \overset{\star}{\in} \{0, 1\}$ describes a symbolic parameter that represents neither $\mathfrak{a} = 0$ nor $\mathfrak{a} = 1$ but rather refers to the abstract un-evaluated symbol itself. In the simplest terms, the former acts as a temporary placeholder whereas the latter is, by design, kept symbolic. Expressed another way, consider $r \in \mathbb{R}$ and $\mathfrak{r} \overset{\star}{\in} \mathbb{R}$. Here, r refers to a particular (albeit undisclosed) real value, while \mathfrak{r} refers to the general concept of a real variable.

An abstract parameter $\mathfrak{p} \overset{\star}{\in} P$ may later be instantiated to a specific value from the set P like so: $\mathfrak{p} \rightarrow p$, where $p \in P$. For instance, if $P = \{1, 2, 3, 4\}$, the abstract parameter \mathfrak{p} may later be instantiated to $\mathfrak{p} = 2$. Thereafter, \mathfrak{p} is no longer an abstract parameter, but rather a specific value, namely 2.

Given lemma 1, which explicitly describes what each element of a function refers to, the relationship between concrete variables, such as $a \in \mathbb{B}$, and abstract variables, such as $\mathfrak{a} \overset{\star}{\in} \mathbb{B}$, may be expressed as follows:

$$\mathfrak{a}(a) = a \tag{2}$$

Lemma 1. *Consider the following:*

$$\phi = \Phi(a) = a\pi \tag{3}$$

where $a \in \{0, 1\}$. Here:

- ϕ refers to the output of the function Φ for a specific value of a . It hence refers to a particular value, $\phi \in \{0, \pi\}$.
- Φ refers to the function itself, described as $\Phi(a) = a\pi$. It can likewise be interpreted to refer to the general expression $a\pi$, for a symbolic, non-specific a . It does not refer to a specific value, though its set of possible outcomes is known: $\text{Image}(\Phi) = \{0, \pi\}$.
- $\Phi(a)$ refers to the function Φ evaluated for a particular input $a \in \{0, 1\}$. For all intents and purposes, to what it refers is the same as that of ϕ . Hence, $\Phi(a) \in \{0, \pi\}$.
- $a\pi$ describes an algebraic expression. Depending on the context, it can refer to either the expression for a specific value of a or generically for a general symbolic a .

In typical circumstances, these precise distinctions are pedantic and superfluous. However, in parts of this thesis (particularly chapter 3) these distinctions are important and this precision is necessary. Indeed, even further clarity can usually be garnered from context. Nevertheless, to help reduce ambiguity, this thesis introduces and generally — unless stated otherwise — adheres to a convention whereby abstract parameters are written in *Fraktur* font:

$$\begin{aligned} \alpha, \beta, \gamma, \dots &\in \mathbb{R}, \\ a, b, c, \dots &\in \mathbb{B}, \\ \mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \dots &\overset{*}{\in} \mathbb{B}. \end{aligned} \tag{4}$$

A circuit, ZX-diagram, or expression containing at least one abstract parameter is said to be *parameterised*, and the word *static* is used in this thesis to mean non-parameterised.

Furthermore, this thesis (particularly chapter 4) utilises a number of mathematical struc-

tures — often in a programmatic context — and conversions between them. For clarity, except where stated otherwise, a tensor (see definition 4) is treated as an n -dimensional array (with integer $n \geq 0$) and is sometimes notated with an overset arrow. Where used, this notation emphasises that, within the relevant context, a multi-dimensional tensor is merely a useful conceptual abstraction of what, in software, is stored as a linear vector. Consistent with programmatic notation, $\vec{T}[2, 5, 3]$ refers to the element at index $(2, 5, 3)$ of tensor \vec{T} . Likewise, given a vector $\vec{a} = (2, 5, 3)$, the equivalent element may be retrieved by $\vec{T}[\vec{a}]$. To this end, a set of abstract parameters, such as $S = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\}$, may be converted into a vector, $\vec{S} = (\mathfrak{a}, \mathfrak{b}, \mathfrak{c})$, (maintaining the order shown in the set) with the notation: $\vec{S} := [S]$. Moreover, if multiple sets or vectors share a common abstract parameter, then its instantiation to a concrete value should, unless stated otherwise, be considered to update every reference to it accordingly.

The following is a summary of the obscure and original terms and operators, together with the pages on which they are defined:

Terms and Operators		
Symbol	Term / Operator	Defined
$\text{Image}(x)$	Image	Page iv
$\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \dots$	Abstract parameters	Page iv
$\overset{\star}{\in}$	Parameterised membership	Page iv
	Staticity	Page vi
\approx	Equal up to a scalar	Page 22
\cong_{struct}	Structural isomorphism	Page 22
$\overset{\epsilon}{\approx}$	Equal up to an error	Page 33
	Parametric symmetry	Page 81
	Phase polarisation	Page 84
\oplus	XOR	Page 84

Table of Contents

1	Introduction	1
2	Background	4
2.1	Quantum Computing	4
2.1.1	Quantum States	5
2.1.2	Quantum Gates	11
2.1.3	Quantum Circuits	16
2.2	ZX-Calculus	19
2.2.1	Introduction to ZX-Calculus	20
2.2.2	Rewriting Strategies	24
2.2.3	Circuit Classes	28
2.3	Classical Simulation	31
2.3.1	Strong and Weak Simulation	33
2.3.2	Marginal Probabilities	35
2.3.3	Tensor Contraction	37
2.3.4	Stabiliser Decomposition	40
2.4	Hypergraph Partitioning	55
2.4.1	Weighted Hypergraphs	55
2.4.2	Minimum Balanced k -Cut	57
2.4.3	Partitioning ZX-Diagrams	58
2.5	GPU Parallelism	63
2.5.1	Parallel Processing	64
2.5.2	GPU Architecture	69
2.5.3	GPU Memory Structure	72
2.5.4	Data Coalescing	74

2.5.5	Data Pipelining	78
3	Parameterised ZX-Calculus	80
3.1	Parametric Symmetry	81
3.2	Parameterising ZX-Calculus	83
3.2.1	Polarising Phases	84
3.2.2	Parameterising State Copy	85
3.2.3	Parameterising the Remaining Rewriting Rules	88
3.2.4	Parameterised Scalar Expressions	96
3.3	GPU-Parallelised Evaluation	102
3.3.1	Condensing the Data Structure	103
3.3.2	Further Considerations	106
3.3.3	Computing the Subterms	108
3.3.4	Computing the Terms	119
3.3.5	Parallelised Summation Algorithm	122
3.3.6	Summary of the New Method	127
3.4	Application to Classical Simulation	132
3.4.1	Repeated Strong Simulation	132
3.4.2	Computing Individual Marginal Probabilities	132
3.4.3	Repeated Weak Simulation	133
3.5	Results	135
3.5.1	Experimental Setup	135
3.5.2	Experimental Measurements	136
3.6	Further Applications	141
3.6.1	Application to Circuit Measurements	141
3.6.2	Parameterising Stabiliser Decompositions	147
3.7	Conclusions	148

4	Smarter ZX-Diagram Partitioning	151
4.1	Formalising the Existing Method	151
4.2	Redundancy Mitigation via Parameterisation	154
4.3	GPU-Parallelised Cutting	161
4.4	Pairwise Partition Regrouping	162
4.5	The ZX-Partitioner	174
4.6	Estimating Runtime	181
4.6.1	Direct Decomposition	181
4.6.2	Smart Partitioning	182
4.7	Results	186
4.8	Tensor Contraction and Compound Circuits	192
4.9	Improving Partitionability	195
4.10	Conclusions	199
5	Procedurally Optimised ZX-Diagram Cutting	201
5.1	Efficient Graph Cutting	202
5.1.1	Slicing Spider Sandwiches	203
5.1.2	CNOT Grouping	205
5.1.3	Cutting in Tiered Structures	208
5.1.4	Cut Order Correction	210
5.2	Optimised Cutting Procedure	212
5.3	Results	215
5.3.1	Circuit Generation	215
5.3.2	Complexity and Efficiency	216
5.3.3	Experimental Measurements for Random Circuits	217
5.4	Conclusions	221
6	Dynamic T-Decomposition	224

6.1	Deriving Dynamic Decompositions	225
6.2	Results	232
6.3	Conclusions	239
7	Conclusions and Future Directions	241
	Bibliography	244

1 | Introduction

Classical simulation is a vital tool for understanding, verifying, and analysing quantum computations, and for benchmarking and quantifying their quantum advantage, particularly as quantum hardware remains in its early and limited stages. Although small and simple quantum systems are efficiently simulable with classical hardware, the same is not true for larger, more practical quantum systems. Simulating such large quantum algorithms classically is a notoriously inefficient task, with runtimes that grow exponentially with the size and/or complexity of the system.

Stabiliser decomposition and tensor contraction are two effective methods by which quantum circuits are classically simulated — both with their advantages and disadvantages and both the subject of much research and improvement in recent years. Notably, the graphical language of ZX-calculus has been extensively applied in aid of this research, particularly for improving the efficiency of stabiliser decomposition.

It is in this context that this thesis originates. Its aim is to present novel techniques, utilising ZX-calculus, by which quantum circuits may be classically simulated with greater efficiency than the existing approaches. Significantly, with few exceptions, most of the prior literature using ZX-calculus for classical simulation has focused on finding new, more efficient stabiliser decompositions. The work presented in this thesis, on the other hand, deviates from this tradition, instead focusing on offering new techniques altogether for improving the efficiency of classical simulation of quantum circuits.

The subsequent chapters may be summarised as follows:

Chapter 2 covers the relevant technical background of quantum comput-

ing and GPU parallelism and provides an introduction and overview of ZX-calculus. This chapter also contains an introduction to, and literature review of, both classical simulation of quantum circuits and hypergraph partitioning. This includes an exploration and comparison of the prominent techniques presently used to solve these problems, together with their benefits and limitations.

Chapter 3 presents a generalisation of the rewriting rules of ZX-calculus to support appropriately restricted parameterised phases. It is subsequently demonstrated how this may be used to compute much of the work involved in classical simulation in parallel, particularly making use of the GPU. Ultimately, it is shown that, even with commercial hardware, this offers a significant linear speedup, sometimes beyond a factor of 100.

Chapter 4 greatly extends existing work on partitioning ZX-diagrams to aid in classical simulation, taking advantage of known techniques in the graph theory literature. Primarily, this chapter details a new *hybrid* method which leverages the advantages of both stabiliser decomposition and tensor contraction to significantly improve the efficiency with which quantum circuits (particularly those with more localised multi-qubit gates) may be classically simulated. The results include circuits for which classical simulation is infeasible via the existing stabiliser decomposition and tensor contraction approaches, but which may be simulated within seconds via this hybrid method.

Chapter 5 challenges the assumption, implicit in the literature, that stabiliser decomposition methods of classical simulation are improved solely by finding new decompositions of higher efficiencies. To this end, this chapter demonstrates that well-motivated applications of lower efficiency

decompositions can in fact lead to a greater *overall* efficiency after ZX-calculus simplification is taken into account. This is exemplified with a heuristic method based on propagating weights to determine efficient patterns in which to apply the trivial ‘cutting’ decomposition. Lastly, it is shown that this approach enables classical simulation with a computational complexity of $O(2^{0.127t})$, as compared to $O(2^{0.196t})$ for the preceding state of the art stabiliser decomposition approach, given a circuit with t ‘non-Clifford’ gates.

Chapter 6 utilises the principles of chapter 5 to derive new stabiliser decompositions based on heuristically motivated applications of the trivial ‘cutting’ decomposition. The resulting decompositions take advantage of patterns common to quantum circuits expressed as reduced ZX-diagrams and, unlike typical decompositions in the literature, are dynamic and scalable in their structure. The results highlight exponential speedups for the task of classical simulation, versus the use of existing alternative decompositions, with the extent of this speedup dependent on the class of circuit considered.

Chapter 7 closes the thesis with a summary of the novel work introduced and the conclusions that may be drawn from their results. This puts into context the contribution this work offers to the literature and broadly highlights potential future work that may extend from this thesis.

2 | Background

2.1 Quantum Computing

As Moore’s Law (Schaller, 1997; Mack, 2011) — the observation that classical computing power doubles approximately every two years — approaches physical limits due to transistor miniaturisation challenges (Theis & Wong, 2017), quantum computing has begun to emerge as a promising alternative for sustaining computational growth. This represents a paradigm shift in computation, leveraging the strange phenomena of quantum mechanics to process information in ways impossible with classical computers. With quantum bits that may exist in a superposition of states, together with the entanglement of such ‘*qubits*’, massively parallel computations are possible, beyond the scope of even theoretical classical computation (Hey, 1999).

While still in its infancy, research into quantum computing has highlighted significant potential (and dangers), with the design of quantum algorithms such as the famous Grover’s algorithm (Grover, 1996), providing a quadratic speedup for unstructured search problems against the best possible classical algorithms. In a similar vein is Shor’s algorithm (Shor, 1999) for efficiently factoring large integers into prime numbers in polynomial time — a problem that is exponentially slower to solve classically. This has profound implications for cryptography, as it threatens the security of the *RSA* encryption protocol (Rivest, Shamir, & Adleman, 1978) upon which much of internet security currently relies (Boneh et al., 1999), and which is dependent upon the presumed hardness of this problem.

There have likewise been many developments in quantum *hardware* in recent years, giving rise to *Noisy Intermediate-Scale Quantum (NISQ)* devices (Preskill,

2018). Due to the technical difficulties of developing quantum computers, these NISQ devices are limited to only tens or hundreds of qubits and are highly susceptible to noise and errors (Lau, Lim, Shrotriya, & Kwek, 2022). Nevertheless, despite not yet achieving full fault-tolerant quantum computing, these devices have proven useful in exploring and demonstrating the potential advantages of quantum algorithms in various fields, from quantum chemistry (Bharti et al., 2022) to combinatorial optimisation problems (Blekos et al., 2024).

2.1.1 Quantum States

In classical computing, the smallest unit of data is a binary digit (*'bit'*), existing, at any given time, in one of two possible states: 0 or 1. Information is processed on a classical computer by manipulating bits using logic gates, which map one or more input bits to a single output bit, according to simple rules. These gates can be expressed using truth tables, which show the output bit corresponding to every possible input bitstring. Composing such gates together, with the outputs of some entering the inputs of others, one may construct a Turing-complete (Turing, 1936) complex network of logical operations, known as a logic circuit (Groote, Morel, Schmaltz, & Watkins, 2021).

Conversely, the fundamental unit of information in a quantum computer is a quantum bit (*'qubit'*). Whereas a classical bit (in random access memory) is physically stored as an electrical charge, or lack thereof, in a capacitor, a qubit is physically realised using quantum systems, such as the spin of an electron (Harneit, 2002; Wesenberg et al., 2009), the polarisation of a photon (Slussarenko & Pryde, 2019; Takeda & Furusawa, 2019), or the energy states of a superconductor (H.-L. Huang, Wu, Fan, & Zhu, 2020; Gambetta, Chow, & Steffen, 2017). As a result, qubits are subject to, and may take advantage of, the phenomena of quantum mechanics. This is what enables quantum computers to perform tasks which are practically

incomputable¹ with a classical computer, giving rise to the *quantum advantage*.

As with a classical bit, a qubit may be in the 0 state or 1 state. In the context of qubits, these are known as the computational basis states (Nielsen & Chuang, 2010) and are denoted $|0\rangle$ and $|1\rangle$ respectively. Here, $|\psi\rangle$ denotes a quantum state in Dirac notation (otherwise known as ‘*Bra-Ket notation*’) (Dirac, 1939). Unlike a classical bit, however, a qubit may exist in a *superposition* of these basis states:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \quad (2.1)$$

where $\alpha, \beta \in \mathbb{C}$ are *probability amplitudes*, satisfying the normalisation condition $|\alpha|^2 + |\beta|^2 = 1$. Upon measurement, the qubit will collapse to the state $|0\rangle$ with probability $|\alpha|^2$ or to the state $|1\rangle$ with probability $|\beta|^2$.

Any such quantum state may be represented as a point on the surface of the *Bloch sphere* (Nielsen & Chuang, 2010), where the poles denote the classical basis states, $|0\rangle$ and $|1\rangle$. This is illustrated in figure 2.1.

A more compact notation sees equation 2.1 expressed as a column vector:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad (2.2)$$

with:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.3)$$

¹Any Turing-complete classical computer can theoretically compute anything a quantum computer can. However, *quantum advantage* tasks may *necessarily* require exponentially more time (and/or space) to compute classically, meaning quantum computers can offer polynomial speedups against even the theoretically most efficient classical algorithms for such tasks.

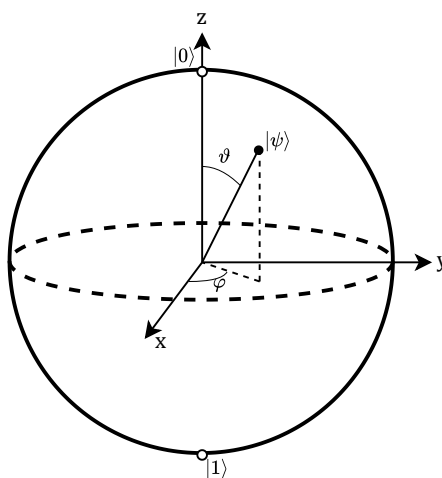


Figure 2.1: The Bloch sphere, illustrating the state of a qubit with two angles, $|\psi\rangle = \cos\left(\frac{\vartheta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\vartheta}{2}\right)|1\rangle$.

The Hermitian adjoint (conjugate transpose) of such a ‘ket’ is then expressed as a ‘bra’:

$$\langle\psi| = \begin{bmatrix} \alpha^* & \beta^* \end{bmatrix} \quad (2.4)$$

where α^* and β^* are the complex conjugates of α and β respectively.

Two parallel qubit states may be expressed as a single column vector by taking their tensor product (\otimes):

$$|\psi\phi\rangle := |\psi\rangle \otimes |\phi\rangle \quad (2.5)$$

such that the 2-qubit basis states are:

$$\begin{aligned}
|00\rangle &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} & |01\rangle &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} & |10\rangle &= \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & |11\rangle &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\end{aligned} \tag{2.6}$$

Hence, two general and parallel single-qubit states:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad |\phi\rangle = \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \tag{2.7}$$

may be expressed as:

$$|\psi\phi\rangle = \begin{bmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{bmatrix} \tag{2.8}$$

Returning to Dirac notation, this represents a linear combination of the 2-qubit basis states:

$$|\psi\phi\rangle = \alpha\gamma |00\rangle + \alpha\delta |01\rangle + \beta\gamma |10\rangle + \beta\delta |11\rangle \tag{2.9}$$

with $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ and $|\alpha\gamma|^2 + |\alpha\delta|^2 + |\beta\gamma|^2 + |\beta\delta|^2 = 1$.

Following this logic, any n -qubit state may be expressed as a length 2^n column vector. This may be interpreted as a vectorisation of a uniform rank n tensor of length 2:

Definition 4. A tensor is an n -dimensional array with integer $n \geq 0$ (and hence

a generalisation of an $n = 0$ scalar, $n = 1$ vector, and $n = 2$ matrix, supporting higher dimensions).

Remark 1. The number of dimensions (or axes), n , of a tensor is known as its **rank**.

Given dimension lengths $m_1, m_2, \dots, m_n \in \mathbb{N}$, a rank n tensor, T , with (for example) complex elements, is given by:

$$T \in \mathbb{C}^{m_1 \times m_2 \times \dots \times m_n} \quad (2.10)$$

Any element of the tensor may be indexed by specifying its Cartesian position in each dimension, as such:

$$T_{i_1, i_2, \dots, i_n} \in \mathbb{C} \quad (2.11)$$

with $i_j \in \{1, 2, \dots, m_j\} \forall j$.

A tensor with a consistent length, $m \in \mathbb{N}$, in each dimension, $m_j = m \forall j$, is said to be **uniform** and its shape may be fully described by its rank n and length m , giving a total of m^n elements.

In addition to superposition, qubits may also exhibit *entanglement*, whereby the quantum states of two or more qubits interact to become correlated. This means the state of one qubit depends upon the state of another (regardless of the physical distance between them). The most famous example is the following *Bell state* (Nielsen & Chuang, 2010):

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle) \quad (2.12)$$

This demonstrates an entanglement of two qubits. If one qubit is measured and its state collapsed to $|0\rangle$ then the other qubit — regardless of its distance from the first — will likewise instantly collapse to the same state, $|0\rangle$. Similarly, if the first qubit is measured and collapsed to $|1\rangle$, the second qubit will also instantly collapse to $|1\rangle$, emphasising the instantaneous correlation between the qubits.

A final point to note is that quantum states are very delicate. When a quantum system is in a superposition, its components (probability amplitudes) may interfere with one another in a phenomenon known as *quantum interference*. Small environmental noise, such as heat and electromagnetic radiation (Zurek, 2003; Breuer & Petruccione, 2002), can induce small fluctuations in the quantum state (impacting its probability amplitudes), leading to *decoherence*. This may also result in a superposition collapsing prematurely.

Decoherence is a significant problem in the field of quantum computing and poses practical limitations on the scale and accuracy of quantum circuits which may be computed with NISQ hardware. Efforts to improve quantum hardware against decoherence are actively underway (De Leon et al., 2021). One key area of research focuses on enhancing qubit coherence times by developing more stable qubit technologies, such as superconducting qubits (Kjaergaard et al., 2020; Devoret, Wallraff, & Martinis, 2004), trapped ions (Häffner, Roos, & Blatt, 2008; Bruzewicz, Chiaverini, McConnell, & Sage, 2019; Georgescu, 2020), and topological qubits (Freedman, Kitaev, Larsen, & Wang, 2003), which are less susceptible to environmental noise.

Additionally, there has been much research into *software* improvements to mitigate the effects of quantum decoherence (Roffe, 2019; Lidar & Brun, 2013). This is known as ‘*Quantum Error Correction*’ (*QEC*) and involves various techniques designed to protect quantum information from errors due to decoherence. QEC

relies on encoding quantum information across multiple physical qubits to form logical qubits. These logical qubits can then detect and correct errors without directly measuring the quantum state, preserving the delicate superpositions and entanglements. However, implementing QEC requires a large overhead in terms of the number of physical qubits and gates, which makes it particularly challenging on current NISQ devices. Despite this, QEC is considered essential for achieving fault-tolerant quantum computing at scale (Gottesman, 2002).

2.1.2 Quantum Gates

Conventionally, quantum algorithms are expressed in *quantum circuit notation*, analogous to the traditional logic circuit notation of classical (binary) computing. In this framework, qubits are represented as horizontal wires, with time progressing from left to right. Quantum gates are shown as operations acting on one or more qubits, mapping their input state (entering the left) to an output state (exiting the right) (Nielsen & Chuang, 2010).

A classical logic gate maps one or two binary inputs to a single binary output² and its behaviour is expressed with a truth table showing every input to output mapping. Conversely, a quantum logic gate is reversible and maps $n \geq 1$ qubit inputs to n qubit outputs, and its behaviour is expressed as a unitary $2^n \times 2^n$ matrix (or indeed a uniform rank $2n$ tensor of length 2 if not vectorised).

For example, the *quantum NOT* gate (also known as the *Pauli-X* gate), depicted in circuit notation in either of the following forms:

$$\text{---} \oplus \text{---} \quad \equiv \quad \text{---} \boxed{X} \text{---}$$

is defined by the following matrix:

²This is true at least of the seven basic logic gates: AND, OR, XOR, NOT, NAND, NOR, and XNOR.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (2.13)$$

such that, applied to the classical basis states, it acts just as the classical NOT gate:

$$\begin{aligned} X |0\rangle &= |1\rangle \\ X |1\rangle &= |0\rangle \end{aligned} \quad (2.14)$$

The Pauli-X gate may be interpreted as inducing a π radian rotation around the X-axis of the Bloch sphere. This gate belongs to the *Pauli* family of gates, together with the Pauli-Y and Pauli-Z gates:

$$\text{---} \boxed{Y} \text{---} \quad \text{---} \boxed{Z} \text{---}$$

which respectively act as π radian rotations about the Y- and Z- axes:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (2.15)$$

Note that the Pauli gates are often alternatively denoted as σ_x , σ_y , and σ_z .

Another important gate is the *Hadamard gate*:

$$\text{---} \boxed{H} \text{---}$$

This is defined by the following matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.16)$$

such that, applied to either basis state this produces an equal superposition of both:

$$\begin{aligned}
H|0\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \equiv |+\rangle \\
H|1\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \equiv |-\rangle
\end{aligned}
\tag{2.17}$$

This also defines the $|+\rangle$ and $|-\rangle$ states, known as the *Hadamard* (or *diagonal*) basis states:

$$|+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad |-\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}
\tag{2.18}$$

The Hadamard gate can be interpreted as inducing a $\frac{\pi}{2}$ radian rotation around the Y-axis of the Bloch sphere, followed by a π radian rotation about the X-axis: $H = XY^{1/2}$.

Two more single-qubit gates of importance are the S and T gates:

$$\text{---} \boxed{S} \text{---} \quad \text{---} \boxed{T} \text{---}$$

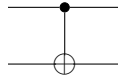
These are given by the following matrices:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}
\tag{2.19}$$

These act as Bloch sphere rotations about the Z-axis of $\frac{\pi}{2}$ radians and $\frac{\pi}{4}$ radians, respectively. As such, $S = T^2$. Furthermore, the adjoint (or conjugate transpose) of these, S^\dagger and T^\dagger , act as Z-axis rotations of $-\frac{\pi}{2}$ and $-\frac{\pi}{4}$, respectively. (Note that $S^\dagger = S^3$ and $T^\dagger = T^7 = S^3T$.)

In addition to these single-qubit gates, there are a handful multi-qubit gates important to this thesis. In particular, the *controlled-NOT*, or *CNOT*, gate (otherwise known as the *controlled-X* or *CX* gate) acts upon two qubits, labelled the *control*

qubit and the *target* qubit:

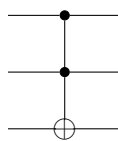


where the smaller black circle denotes the control qubit.

This applies the quantum NOT gate (equation 2.13) to the target qubit if the control qubit is in the state $|1\rangle$ and does nothing (applies the identity) if the control qubit is in the state $|0\rangle$. If the control qubit is in a superposition state and is not already entangled with the target qubit, and the target qubit is not in the $|+\rangle$ or $|-\rangle$ state, then the CNOT gate will create an entanglement between the two qubits. The corresponding matrix of the CNOT gate follows:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (2.20)$$

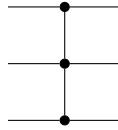
Next, there is the *controlled-controlled-NOT*, or *CCNOT*, gate (otherwise known as the *Toffoli* gate or *CCX* gate) (Aharonov, 2003):



This applies the quantum NOT gate to the target qubit when *both* control qubits are in the $|1\rangle$ state. Its matrix follows:

$$\text{Toff} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (2.21)$$

Similarly, the *controlled-controlled-Z*, or *CCZ*, gate:



(in the computational basis) applies the Z gate to the target qubit, conditional on two control qubits. Interestingly, as one may observe from its depiction above, the CCZ gate is symmetrical, meaning its behaviour remains unchanged when the target qubit is swapped with either control qubit. The matrix representing the CCZ gate is as follows:

$$CCZ = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \quad (2.22)$$

Lastly, while trivial, it is worth noting that the wire itself induces no effect, essentially mapping any quantum state to itself. It is hence equivalent to the identity matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2.23)$$

and is sometimes expressed as a gate itself (namely the ‘*identity gate*’):



With this in mind, the *SWAP* gate, which unsurprisingly swaps the states of two qubits³, may be visualised as two wires switching sides:



This gate is represented by the following matrix:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.24)$$

2.1.3 Quantum Circuits

Just as classical logic gates may be composed together to produce complex logic circuits, being Turing-complete, so too may *quantum* gates be composed to construct *quantum* circuits, capable of implementing any quantum algorithm (Nielsen

³On the topic of moving a quantum state from one qubit to another, it should be noted that — due to the *no-cloning theorem* (Wootters & Zurek, 1982) — it is not possible to *copy* an arbitrary unknown quantum state to another qubit.

& Chuang, 2010).

Two quantum gates, A and B , of n and m qubits respectively, may be composed in parallel by taking their tensor (or *Kronecker*) product (Coecke & Kissinger, 2018), $A \otimes B$:

$$\begin{array}{c} \left(\begin{array}{c} \text{---} \boxed{A} \text{---} \\ \otimes \\ \text{---} \boxed{B} \text{---} \end{array} \right) = \begin{array}{c} \text{---} \boxed{A} \text{---} \\ \text{---} \boxed{B} \text{---} \end{array} \equiv \begin{array}{c} \text{---} \boxed{A \otimes B} \text{---} \end{array} \end{array} \quad (2.25)$$

In this case, $n = m = 1$. The result, $(A \otimes B)$, may be treated as a single $(n + m)$ -qubit quantum gate and may be composed further with other gates if desired.

Two quantum gates, C and D , may also be composed *serially*, provided they act on the same number n of qubits, by taking their matrix product, $D \circ C$:

$$\begin{array}{c} \left(\begin{array}{c} \text{---} \boxed{D} \text{---} \\ \circ \\ \text{---} \boxed{C} \text{---} \end{array} \right) = \begin{array}{c} \text{---} \boxed{C} \text{---} \boxed{D} \text{---} \end{array} \equiv \begin{array}{c} \text{---} \boxed{D \circ C} \text{---} \end{array} \end{array} \quad (2.26)$$

In this illustration, $n = 1$. As with parallel composition, the result of serial composition may be treated as a single quantum gate, acting on n qubits.

In this way, large and complex quantum circuits may be constructed, as the following example demonstrates:

$$\begin{aligned}
& \begin{array}{c} \text{---} \boxed{A} \text{---} \boxed{C} \text{---} \boxed{D} \text{---} \\ \text{---} \boxed{B} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{A \otimes B} \text{---} \boxed{D \circ C} \text{---} \\ \text{---} \boxed{I} \text{---} \end{array} \quad (2.27) \\
& = \begin{array}{c} \text{---} \boxed{A \otimes B} \text{---} \boxed{(D \circ C) \otimes I} \text{---} \\ \text{---} \end{array} \\
& = \begin{array}{c} \text{---} \boxed{((D \circ C) \otimes I) \circ (A \otimes B)} \text{---} \\ \text{---} \end{array}
\end{aligned}$$

Here, the four single-qubit gates — each expressible as a 2×2 matrix — have, through applications of matrix multiplication and tensor products, been compounded into a single gate expressible as a 4×4 matrix.

Moreover, regarding serial composition, when there is not a one-to-one correspondence between the outputs of one gate and the inputs of the next, one of the gates may be composed in parallel with the identity gate (being just a wire), as the above example highlights.

In fact, with only a very small handful of gates as ‘*building block*’, all higher level gates and indeed any quantum computation may be constructed. More precisely, such a universal gateset (of which there are many options) may approximate any unitary operation, and hence any quantum algorithm, to arbitrary precision (Deutsch, Barenco, & Ekert, 1995). For instance, the $\{H, CNOT, T\}$ gateset is universal (Forest, Gosset, Kliuchnikov, & McKinnon, 2015).

Circuits comprised exclusively of the ‘*Clifford*’ gateset, $\{H, S, CNOT\}$, (plus implicitly the identity gate) are in fact efficiently simulable classically:

Theorem 1 (The Gottesman-Knill Theorem). *Any quantum computation that consists solely of Clifford gates and Pauli measurements can be efficiently simulated*

on a classical computer (Gottesman, 1998; Aaronson & Gottesman, 2004).

Therefore, the Clifford gateset is a very significant one, though it is not universal as these gates cannot approximate arbitrary unitary transformations. However, introducing the T-gate produces the universal gateset: $\{H, S, CNOT, T\}$ ⁴ (Kliuchnikov, 2013; Forest et al., 2015). This is known as the *Clifford+T* gateset.

Using this gateset, the *SWAP* gate may be expressed:

$$\begin{array}{c} \diagup \diagdown \\ \diagdown \diagup \end{array} = \begin{array}{c} \bullet \oplus \bullet \\ \oplus \bullet \oplus \end{array} \quad (2.28)$$

and the CCZ gate:

$$\begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \equiv \begin{array}{c} \bullet \quad \bullet \quad \bullet \\ \bullet \quad \bullet \quad \bullet \\ \oplus \quad T^\dagger \quad T \quad \oplus \quad T^\dagger \quad T \end{array} \quad (2.29)$$

with the Toffoli gate following from the above:

$$\begin{array}{c} \bullet \\ \bullet \\ \oplus \end{array} \equiv \begin{array}{c} \bullet \quad \bullet \quad \bullet \\ \bullet \quad \bullet \quad \bullet \\ \oplus \quad H \quad T^\dagger \quad \oplus \quad T \quad \oplus \quad T^\dagger \quad \oplus \quad T \quad H \end{array} \quad (2.30)$$

2.2 ZX-Calculus

The ZX-calculus (Coecke & Kissinger, 2018; van de Wetering, 2020; Kissinger & van de Wetering, 2024) is a graphical language for expressing and manipulating quantum computations, originating in 2008 in the work of (Coecke & Duncan,

⁴This gateset is universal but not minimal, as $S = T^2$, making the S gate redundant. However, with the significance of the Clifford gateset, the use of the Clifford+T set is very convenient.

2008, 2011). This language sees quantum circuits abstracted to *ZX-diagrams*, which may be simplified and reasoned upon via the known *rewriting rules*. It has been used extensively in a variety of quantum computing problems, including circuit optimisation (Kissinger & van de Wetering, 2020b; Cowtan, Dilkes, Duncan, Simmons, & Sivarajah, 2019; De Beaudrap, Bian, & Wang, 2019; Duncan, Kissinger, Perdrix, & Van De Wetering, 2020), quantum error correction (Carette, Horsman, & Perdrix, 2019; Chancellor, Kissinger, Zohren, Roffe, & Horsman, 2023; Duncan & Lucas, 2013; Garvie & Duncan, 2017), solving #SAT problems (de Beaudrap, Kissinger, & Meichanetzidis, 2020; Laakkonen, 2022; Laakkonen, Meichanetzidis, & van de Wetering, 2022, 2023), natural language processing (Kartsaklis et al., 2021; Coecke, de Felice, Meichanetzidis, & Toumi, 2020; Coecke, de Felice, Meichanetzidis, Toumi, Gogioso, & Chiappori, 2020; Toumi, 2022), lattice surgery (de Beaudrap & Horsman, 2020; de Beaudrap, Duncan, Horsman, & Perdrix, 2019; Gidney & Fowler, 2019; Hanks, Estarellas, Munro, & Nemoto, 2020), measurement-based quantum computing (Duncan & Perdrix, 2010; Stollenwerk & Hadfield, 2024; Schwetz & Noack, 2024), and — most relevant to this thesis — classical simulation (Kissinger & van de Wetering, 2022; Kissinger, van de Wetering, & Vilmart, 2022; Koch, Yeung, & Wang, 2023; Codsì, 2022).

2.2.1 Introduction to ZX-Calculus

The building blocks of ZX-calculus are vertices known as ‘*spiders*’. Specifically, there exist the green *Z-spider* and the red *X-spider*, being generalisations of the Z- and X- phase gates, respectively. With reference to the basis states in section 2.1.1, these spiders may be defined as follows (Coecke & Kissinger, 2018):

$$\begin{array}{c} \vdots \\ \vdots \end{array} \begin{array}{c} \diagup \\ \diagdown \end{array} \begin{array}{c} \diagdown \\ \diagup \end{array} \begin{array}{c} \vdots \\ \vdots \end{array} \begin{array}{c} \diagup \\ \diagdown \end{array} \begin{array}{c} \vdots \\ \vdots \end{array} \quad := \quad |+\cdots+\rangle \langle +\cdots+| + e^{i\alpha} |-\cdots-\rangle \langle -\cdots-|$$

Special cases of equation 2.31 define the basis states (van de Wetering, 2020):

$$\begin{array}{llll}
 \text{red circle} & = & |+\rangle + |-\rangle & = \sqrt{2} |0\rangle \\
 \text{red circle with } \pi & = & |+\rangle - |-\rangle & = \sqrt{2} |1\rangle \\
 \text{green circle} & = & |0\rangle + |1\rangle & = \sqrt{2} |+\rangle \\
 \text{green circle with } \pi & = & |0\rangle - |1\rangle & = \sqrt{2} |-\rangle
 \end{array}$$

$$\text{---} \square \text{---} \equiv \text{---} \text{---} \text{---} = e^{-i\frac{\pi}{4}} \text{---} (\text{green circle } \frac{\pi}{2}) \text{---} (\text{red circle } \frac{\pi}{2}) \text{---} (\text{green circle } \frac{\pi}{2}) \text{---} \quad (2.33)$$

Just as in quantum circuit notation, spiders may be composed sequentially or in parallel by computing the matrix multiplication or tensor product of their matrices.

Such a composition of spiders is then known as a *ZX-diagram*, a very convenient property of which follows (Coecke & Kissinger, 2018):

Theorem 2. *Isomorphic ZX-diagrams semantically describe the same linear map. Hence, ZX-diagrams may be arbitrarily deformed provided topology is conserved and the inputs and outputs are fixed. This is commonly stated as the meta-rule: **only connectivity matters**.*

Two ZX-diagrams are hence considered equal if they represent the same linear map. Furthermore, it is often convenient to neglect any non-zero constant scalar factor associated with a ZX-diagram⁵. Two ZX-diagrams, A and B , which are equal up to a non-zero constant scalar factor are thus, by convention, denoted: $A \approx B$ (van de Wetering, 2020). For example:

$$\sqrt{2} \begin{array}{c} \text{---} \circ \text{---} \\ | \\ \text{---} \bullet \text{---} \end{array} \approx \begin{array}{c} \text{---} \circ \text{---} \\ | \\ \text{---} \bullet \text{---} \end{array} \quad (2.34)$$

Moreover, there is a broad type of equivalence between ZX-diagrams which concerns its topological structure (and spider and edge types) but ignores phase values:

Definition 5. *In the context of this thesis, two ZX-diagrams, A and B , are said to be ‘structurally isomorphic’, denoted as $A \cong_{\text{struct}} B$, if:*

- *There exists a bijection $f : V_A \rightarrow V_B$ between their respective sets of vertices, V_A and V_B , such that for every vertex $v \in V_A$, the corresponding vertex $f(v) \in V_B$ has the same colour (but not necessarily the same phase),*
- *for every edge (u, v) in A , the corresponding edge $(f(u), f(v))$ exists in B and is of the same type,*

⁵Often, for the sake of brevity, such factors are neglected when ZX-diagrams are written out, but are nevertheless retained in any further calculations where these factors are relevant.

- *for every input/output connected to a vertex u in A , there exists a corresponding input/output connected to $f(u)$ in B , and*
- *the connectivity of the vertices is preserved under this mapping, meaning the graph structure remains identical.*

With these building blocks, it is possible to translate any quantum gate into a ZX-diagram, with some prominent examples following:

$$\begin{array}{llll}
 Z = \text{---} \textcircled{\pi} \text{---} & S = \text{---} \textcircled{\frac{\pi}{2}} \text{---} & CNOT \approx \begin{array}{c} \text{---} \textcircled{\pi} \text{---} \\ | \\ \text{---} \textcircled{\pi} \text{---} \end{array} & CZ \approx \begin{array}{c} \text{---} \textcircled{\pi} \text{---} \\ | \\ \text{---} \textcircled{\pi} \text{---} \end{array} \\
 X = \text{---} \textcircled{\pi} \text{---} & T = \text{---} \textcircled{\frac{\pi}{4}} \text{---} & &
 \end{array} \quad (2.35)$$

Any quantum computation can be expressed as a ZX-diagram (Backens, 2016; Q. Wang, 2022; Jeandel, Perdrix, & Vilmart, 2020), with Clifford and Clifford+T circuits manifesting as follows (Kissinger & van de Wetering, 2024):

Definition 6. A *Clifford ZX-diagram* is a ZX-diagram with spider phases restricted to $n\frac{\pi}{2}$, where $n \in \mathbb{Z}$.

Definition 7. A *Clifford+T ZX-diagram* is a ZX-diagram with spider phases restricted to $n\frac{\pi}{4}$, where $n \in \mathbb{Z}$.

The latter case is sufficient to approximate any quantum computation up to arbitrary precision (Backens, 2016; Q. Wang, 2022). In such ZX-diagrams, a ‘*T-like spider*’ (in certain contexts used interchangeably with ‘*T-spider*’) is a spider with a phase $n\frac{\pi}{4}$ given an odd integer n . (Note that the number of T-gates in a circuit, or T-spiders in a ZX-diagram, is labelled its ‘*T-count*’.)

Moreover, a ZX-diagram with no open input or output wires represents a scalar:

Definition 8. A *scalar ZX-diagram* is a ZX-diagram with no open input or output

wires. Such a diagram is equivalent to a complex scalar.

Furthermore, the adjoint of a ZX-diagram amounts to its horizontal mirroring and inverting the sign of every spider phase, plus a mapping of any scalar coefficient to its complex conjugate. A simple example follows:

$$\left(\begin{array}{c} \text{---} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \right)^\dagger = \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \quad (2.36)$$

2.2.2 Rewriting Strategies

A quintessential component of ZX-calculus is its small set of well-defined ‘*rewriting rules*’. These are equations describing how common structures within ZX-diagrams may be rewritten in a more convenient manner. These rules are summarised in figure 2.2, with their proofs available in (van de Wetering, 2020).

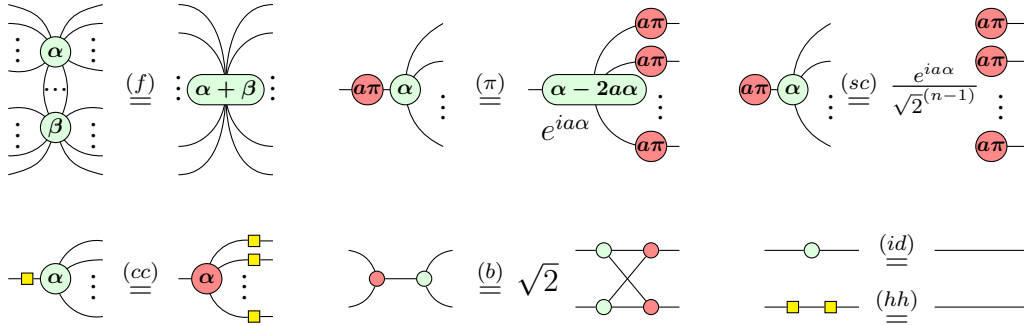


Figure 2.2: The Clifford complete (Backens, 2016) fundamental rewriting rules of ZX-calculus (van de Wetering, 2020), where $\alpha, \beta \in \mathbb{R}$ and $a \in \mathbb{B}$. Meanwhile, n is the number of outgoing edges. These rules are: (f) fusion, (π) π -commutation, (sc) state copy, (cc) colour change, (b) bialgebra, (id) identity removal, and (hh) Hadamard cancellation. These rules remain applicable with all spider colours inverted.

These rewriting rules may be used to simplify ZX-diagrams into smaller and simpler equivalents. For instance, the following *GHZ-state* (Greenberger, Horne,

& Zeilinger, 1989) may be translated into a ZX-diagram and simplified via the rewriting rules (Coecke & Kissinger, 2018):

$$\begin{array}{c}
 |0\rangle \\
 |0\rangle \\
 |0\rangle
 \end{array}
 \begin{array}{c}
 \oplus \\
 \boxed{H} \quad \bullet \\
 \oplus
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \oplus \\
 \oplus \\
 \oplus
 \end{array}
 \equiv
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 =
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 =
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \begin{array}{c}
 \bullet \\
 \bullet \\
 \bullet
 \end{array}
 \quad (2.37)$$

Such simplification can be used to reduce large quantum circuits to smaller, more manageable ZX-diagrams, which in turn may be synthesised back into quantum circuits. This can allow circuits to be expressed with fewer gates, thus minimising decoherence when executed on a quantum computer. Specifically, this approach is very effective at minimising the number of costly T-gates, as detailed in (Kissinger & van de Wetering, 2020b).

The simplification strategy essentially amounts to applying the rewriting rules wherever applicable until no further applications may be made. More precisely, the method, as outlined in (Kissinger & van de Wetering, 2020b) and (Kissinger & van de Wetering, 2022), utilises four broader rules derived from this basic set and shown in figures 2.3. In addition to these rewriting rules are the scalar relations shown in figure 2.4.

These basic rewriting rules of figure 2.2, together with these scalar relations, are alone sufficient to reduce any scalar Clifford ZX-diagram to its scalar, consistent with theorem 1.

On the other hand, scalar *Clifford+T* ZX-diagrams (required for completeness (Backens, 2016; Q. Wang, 2022)) are not fully reducible to scalar via the rewriting rules. Nevertheless, they may undergo (often considerable) simplification.

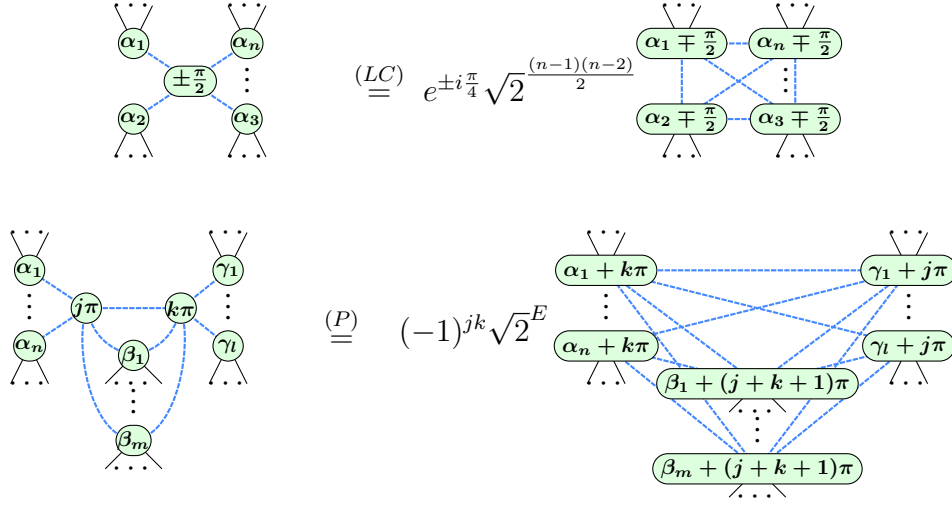


Figure 2.3: The (P) pivoting and (LC) local complementation rules (van de Wetering, 2020), derivable from the basic set of figure 2.2. Here, $\alpha_i, \beta_i, \gamma_i \in \mathbb{R} \forall i$, $l, m, n \in \mathbb{Z}$, $j, k \in \mathbb{B}$, and $E = (n-1)m + (l-1)m + (n-1)(l-1)$.

For simplicity, it is helpful to introduce two new rules, derivable from the basic set of figure 2.2. These are summarised in figure 2.5.

Given these rules, the simplification procedure for reducing scalar diagrams may be summarised as follows (Kissinger & van de Wetering, 2020b):

1. If applicable anywhere in the ZX-diagram, apply LC or P . Repeat until no such instances are found. Every remaining Clifford spider will then take a phase of 0 or π and be connected only to T-like spiders.
2. Apply PG to any remaining Clifford spiders ($j \in \{0, 1\}$).
3. Apply GF wherever applicable. Repeat until no further instances are found.

This procedure will reduce any scalar Clifford ZX-diagram to a scalar result and and scalar *Clifford+T* ZX-diagram to its *reduced gadget form*:

Definition 9. A scalar *Clifford+T* ZX-diagram is in **reduced gadget form** if and

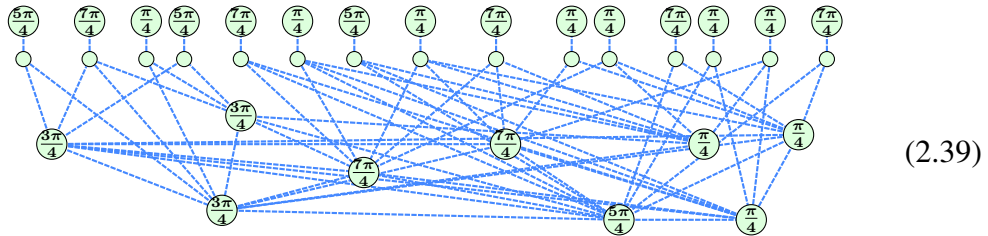
$$\begin{aligned}
\textcircled{\alpha} &= 1 + e^{i\alpha} \\
\textcircled{\alpha} - \textcircled{a\pi} &= \sqrt{2}e^{ia\alpha} \\
\textcircled{\alpha} - \textcircled{\beta} &= \frac{1}{\sqrt{2}} (1 + e^{i\alpha} + e^{i\beta} - e^{i(\alpha+\beta)})
\end{aligned}$$

Figure 2.4: The basic scalar relations of ZX-calculus (van de Wetering, 2020), where $\alpha, \beta \in \mathbb{R}$ and $a \in \mathbb{B}$.

only if every spider is either *T-like* or the root (internal spider) of a **phase gadget**:

$$\textcircled{\alpha} \text{ --- } \textcircled{} \begin{array}{c} \diagup \\ \vdots \\ \diagdown \end{array} \quad (2.38)$$

This procedure takes at most $O(N^3)$ elementary graph operations (vertex deletions, edge toggles, etc.), where N is the number of initial spiders (Kissinger & van de Wetering, 2022), and removes almost all Clifford phases and often many T-like phases. Ultimately, it produces ZX-diagrams in reduced gadget form, which resemble the following example case:



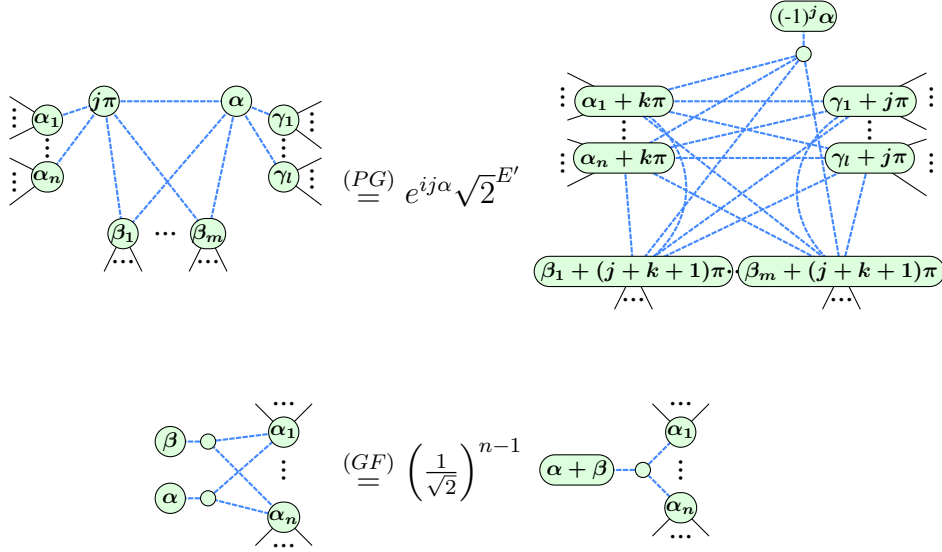


Figure 2.5: The (PG) gadgetisation (or pivoting gadget) rule and the (GF) gadget fusion rule. Both are derivable from the rules of figure 2.2, with $\alpha, \beta \in \mathbb{R}$, $\alpha_i, \beta_i \in \mathbb{R} \forall i, l, m, n \in \mathbb{Z}$, $j, k \in \mathbb{B}$, and $E' = (n - 1)m + lm + (n - 1)l$.

2.2.3 Circuit Classes

There are a number of circuit classes which have been used for benchmarking classical simulation methods in the ZX-calculus literature. Consistent with this literature, this thesis considers randomly generated circuits from the four commonly benchmarked classes:

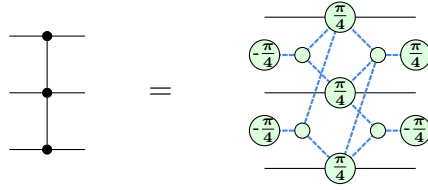
- Clifford+T circuits
- Clifford+T+CCZ circuits
- Instantaneous Quantum Polynomial (IQP) circuits
- Modified hidden shift circuits

In all cases, the ‘initial T-count’ is taken as the number of T-spiders in the ZX-diagram *after* initial Clifford simplification.

Clifford+T circuits are generated with a random placement of gates from the

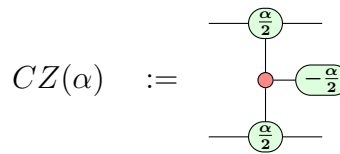
$\{S, HSH, CNOT, T\}$ gateset (Kissinger & van de Wetering, 2019), with an approximately equal distribution of each. The depth and number of qubits are varied to control the T-count and circuit size. A somewhat controlled approach to generating random circuits in this class is via placements of exponentiated Pauli unitaries, as described in (Kissinger & van de Wetering, 2022).

Clifford+T+CCZ circuits (or simply ‘CCZ circuits’) are generated in much the same way, using the gateset of $\{H, S, CNOT, CZ, T, CCZ\}$, with T and CCZ gates each having a 5% sampling probability, and the remaining (Clifford) gates being uniformly sampled. While there are a few ways of translating a CCZ gate into a ZX-diagram, the choice used in this thesis is consistent with the benchmarking literature (Nielsen & Chuang, 2010):

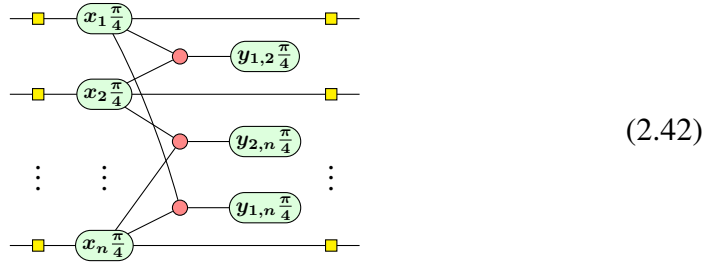

(2.40)

Instantaneous Quantum Polynomial (IQP) circuits were used for benchmarking in (Codsi, 2022; Codsi & van de Wetering, 2022), being considered feasible for quantum hardware, though their hardness for classical simulation and their potential for demonstrating quantum supremacy (Bremner, Montanaro, & Shepherd, 2017) have been debated (Codsi & van de Wetering, 2022).

IQP circuits are composed of Hadamard gates, T^m gates, and $CZ(\frac{k\pi}{2})$ gates, with $m, k \in \mathbb{Z}$ and:

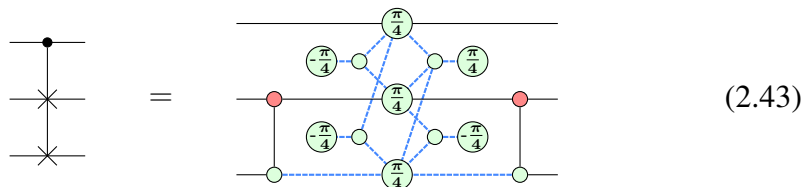

(2.41)

After Clifford simplification, such circuits look akin to the following (Codsì & van de Wetering, 2022):



where $x_i, y_{i,j} \in \mathbb{Z} \forall i, j$.

Lastly, this thesis considers **modified hidden shift circuits**. Hidden shift circuits are designed to solve *hidden shift problems*, where the goal is to find the unknown shift s such that $g(x) = f(x+s)$, given two oracles, f and g . Such circuits (Bravyi & Gosset, 2016) served as benchmarks in a number of related papers (Kissinger & van de Wetering, 2022; Kissinger et al., 2022; Bravyi et al., 2019; Peres & Galvão, 2023), though, as some have argued, subsequent improvements in ZX-calculus simplification has rendered these circuits trivial for classical computation, with conjectures suggesting they are simulable in polynomial time (Codsì, 2022). Consequently, *modified* hidden shift circuits (Koch et al., 2023) have since been used in their place to regain their non-triviality. These circuits are composed of randomly placed Clifford gates plus *Controlled-SWAP* gates (otherwise known as *Fredkin* gates)⁶:



⁶The original hidden shift circuits utilised CCZ gates instead of controlled-SWAP gates. The similarity between the two is easily noticed.

2.3 Classical Simulation

In the present NISQ era of quantum computing, quantum hardware is scarce and limited. Current quantum processors suffer from noise, short coherence times, and gate errors, which restrict their ability to perform deep circuits reliably (Barak & Marwaha, 2021; Bharti et al., 2022). Additionally, the number of available high-quality qubits remains relatively low, making it difficult to execute large-scale quantum algorithms without significant error mitigation. As a result, researchers often rely on classical simulations to test and validate quantum algorithms before running them on quantum devices (or indeed when doing so is infeasible with the current hardware limitations) (Xu, Benjamin, Sun, Yuan, & Zhang, 2023; Vidal, 2003; C. Huang et al., 2020). Ongoing advancements in hardware design (Acharya et al., 2024; J. Wang, Sciarrino, Laing, & Thompson, 2020), error correction (Roffe, 2019; Duncan & Lucas, 2013), and hybrid quantum-classical approaches (Endo, Cai, Benjamin, & Yuan, 2021; McClean, Romero, Babbush, & Aspuru-Guzik, 2016; Callison & Chancellor, 2022) continue to advance the limits of quantum computers, though in the present and near term classical simulation remains an essential tool.

The ability to simulate quantum algorithms with classical hardware has myriad uses, both practical and theoretical, with the following among them:

- **Verification of quantum algorithms:** classical simulation can be used to verify the correctness of quantum algorithms before running them on quantum hardware (Hietala, Rand, Hung, Li, & Hicks, 2020; Liu et al., 2019; Sander, Burgholzer, & Wille, 2024). This can also help with debugging quantum circuits by comparing expected and observed results.
- **Verification of quantum hardware:** classical simulation may also be used

to verify the behaviour and correctness of quantum hardware in much the same manner (Gheorghiu, Kapourniotis, & Kashefi, 2019; Mahadev, 2018).

- **Measuring the quantum advantage:** by computing the same circuit on both classical and quantum hardware, it is possible to quantify the runtime reduction offered by the latter for particular tasks. In this regard, classical simulation can be used to set baselines against which to compare quantum computing performance (C. Huang et al., 2020; Pan & Zhang, 2021).
- **Simulating quantum many-body systems:** modelling quantum systems (Fauseweh, 2024; Jo & Kim, 2022) is an important part of quantum condensed matter physics (Hofstetter & Qin, 2018; Kennes et al., 2021), though with their high noise and limited scale, such modelling on contemporary quantum hardware is often infeasible. However, simulating these models classically allows exact (noiseless) results to be attained.
- **Quantum-classical hybrid algorithms:** hybrid algorithms (McClean et al., 2016; Endo et al., 2021; Callison & Chancellor, 2022) utilise the strengths of both quantum and classical hardware, with the former handling such tasks as exploring large solution spaces or finding optimal solutions to *quantum advantage* problems, while the latter handles such tasks as optimisation and error correction. Hybrid algorithms are heavily utilised in quantum machine learning (Rebentrost, Mohseni, & Lloyd, 2014; Cowlessur, Thapa, Alpcan, & Camtepe, 2024; Zaman, Ahmed, Hanif, Marchisio, & Shafique, 2024), combinatorial optimisation problems (Farhi, Goldstone, & Gutmann, 2014; Blekos et al., 2024; S.-X. Zhang et al., 2022), and cryptography (Scarani et al., 2009; Ren, Wang, & Su, 2022; Mazzoncini, Bauer, Brown, & Alléaume, 2023), and often rely to some degree on classical simulation.
- **Analysing cryptographic security:** some quantum algorithms threaten clas-

sical cryptographic protocols. Famously, Shor’s quantum factorisation algorithm (Shor, 1999) undermines the RSA (Rivest et al., 1978) protocol upon which much of online security presently relies. Classical simulation of such algorithms may provide insights which could help develop stronger, quantum-safe security protocols.

This list is by no means exhaustive, but it serves to highlight the broad utility of classical simulation to the field of quantum computing.

2.3.1 Strong and Weak Simulation

The term ‘*classical simulation*’ in the context of quantum computing may refer to one of two concepts, namely *strong* simulation or *weak* simulation. These are related but distinct ideas, defined as follows (Kissinger & van de Wetering, 2022):

Definition 10. *Strong classical simulation of a quantum circuit is to determine (possibly up to some acceptable margin of error, ϵ) the probability of a particular measurement outcome. More concretely, it is to (either exactly or approximately) determine — with classical hardware — the probability of measuring a particular output bitstring, $(b_1, b_2, \dots, b_n) \in \mathbb{B}^n$, when executing an n -qubit quantum circuit, C , with initial states $(a_1, a_2, \dots, a_n) \in \mathbb{B}^n$:*

$$\left(\frac{1}{\sqrt{2}}\right)^{2n} \begin{array}{c} a_1\pi \\ a_2\pi \\ \vdots \\ a_n\pi \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{C} \\ \boxed{C} \\ \boxed{C} \\ \boxed{C} \end{array} \begin{array}{c} b_1\pi \\ b_2\pi \\ \vdots \\ b_n\pi \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \stackrel{\epsilon}{\approx} \lambda \quad (2.44)$$

where $\lambda \in \mathbb{C}$ and $P(b_1, \dots, b_n | C | a_1, \dots, a_n) = |\lambda|^2$ is the probability in question⁷. Note that $\stackrel{\epsilon}{\approx}$ denotes ‘approximately equal within a margin of error, ϵ ’.

⁷The global scalar factor arises from the states and effects, per equation 2.32.

Definition 11. *Weak classical simulation of a quantum circuit is to probabilistically sample from a circuit's output distribution. In other words, it is to emulate a quantum computation. Concretely, given a particular initial state $(a_1, \dots, a_n) \in \mathbb{B}^n$, produce a randomly selected output bitstring $(b_1, \dots, b_n) \in \mathbb{B}^n$ with a probability consistent with $P(b_1, \dots, b_n | C | a_1, \dots, a_n)$.*

As their names suggest, strong simulation is strictly more powerful than weak simulation. The ability to compute exact (or approximate) probabilities of measurement outcomes allows one to generate samples by drawing from the corresponding probability distribution. Hence:

$$\text{Strong Simulation} \Rightarrow \text{Weak Simulation} \quad (2.45)$$

Naturally, without the quantum advantage, simulating quantum circuits with classical hardware is necessarily, and notoriously, inefficient⁸. Restricted to bits, rather than qubits, classical computers have no efficient means of representing the superposition and entanglement inherent in quantum systems. Nevertheless, any Turing-complete (Turing, 1936) classical computer can, given sufficient finite time, compute any quantum algorithm. However, unlike quantum computers, classical computers require an exponentially growing amount of resources — both time and memory — as the size of the quantum system increases. This exponential scaling is what limits the practical feasibility of classical simulation for large quantum circuits. Thus, while classical simulation of any quantum system is theoretically possible, it becomes infeasible for large cases. This, after all, is what makes quantum computers advantageous in solving certain problems.

There are a variety of approaches to both strong and weak classical simulation,

⁸In fact, this is a conjecture, though one underpinning the entire field of quantum computing.

which differ in how their exponentially growing resource requirements manifest. Some methods scale exponentially with the number of qubits (Jamadagni, Läuchli, & Hempel, 2024; Zulehner & Wille, 2018; Chundury, Li, Suh, & Mueller, 2024), while others scale exponentially with the number of (non-Clifford) gates (Bravyi et al., 2019; Kissinger & van de Wetering, 2022; Kissinger et al., 2022). Others still scale predominantly with the degree of qubit interconnectedness, entanglement, or treewidth of the underlying graph representing the circuit (Wahl & Strelchuk, 2023; Fried et al., 2018; Gray & Kourtis, 2021), though many methods, particularly hybrid approaches (Mandrà, Marshall, Rieffel, & Biswas, 2021; Young, Scese, & Ebne-nasir, 2023), may be impacted by any of all of these to differing extents.

2.3.2 Marginal Probabilities

Most weak simulation techniques are stochastic (Aaronson, 2013; Bremner, Jozsa, & Shepherd, 2011), including Monte Carlo methods (Bravyi, Gosset, & Liu, 2022; Bravyi & Gosset, 2016), which weakly simulate circuits *directly*. However, following from equation 2.45, some methods rely upon firstly computing measurement probabilities via strong simulation. In particular, these methods tend to involve computing many *marginal* probabilities (Kissinger & van de Wetering, 2022; Kissinger et al., 2022), concerning the measurement of specific outcomes for only a subset of qubits:

Definition 12. *A marginal probability is the probability, given an n -qubit circuit with a particular initial state $(a_1, a_2, \dots, a_n) \in \mathbb{B}^n$, of measuring a particular outcome $(b_1, b_2, \dots, b_k) \in \mathbb{B}^k$ on a subset, $k < n$, of the qubits in the circuit, without care of the measurement results on the remaining qubits.*

The most obvious means of computing a marginal probability is to simply sum

over the probabilities of each irrelevant qubit, as per the ‘*summing*’ method of lemma 2 (Kissinger & van de Wetering, 2022).

Lemma 2 (Summing method). *Given an n -qubit circuit, C , with an initial state $(a_1, \dots, a_n) \in \mathbb{B}^n$ the probability of measuring an output $(b_1, \dots, b_k) \in \mathbb{B}^k$ on the first $k < n$ qubits is given by:*

$$P(b_1, \dots, b_k | C | a_1, \dots, a_n) = \frac{1}{4^n} \sum_{c_1, \dots, c_m} \left| \begin{array}{c} a_1 \pi \\ \vdots \\ a_k \pi \\ a_{k+1} \pi \\ \vdots \\ a_n \pi \end{array} \begin{array}{c} \vdots \\ \vdots \\ \vdots \\ \vdots \end{array} \begin{array}{c} \vdots \\ \vdots \\ \vdots \\ \vdots \end{array} \begin{array}{c} b_1 \pi \\ \vdots \\ b_k \pi \\ c_1 \pi \\ \vdots \\ c_m \pi \end{array} \right|^2 \quad (2.46)$$

where $(c_1, \dots, c_m) \in \mathbb{B}^m$ are the outcome measurements on the irrelevant qubits, with $k + m = n$.

The ‘*doubling*’ method, described in lemma 3 (Kissinger & van de Wetering, 2022), provides an alternative approach, where the given circuit, together with its open (irrelevant) qubit outputs, is composed against its own adjoint.

Lemma 3 (Doubling method). *Given an n -qubit circuit, C , with an initial state $(a_1, \dots, a_n) \in \mathbb{B}^n$ the probability of measuring an output $(b_1, \dots, b_k) \in \mathbb{B}^k$ on the first $k < n$ qubits is given by:*

$$P(b_1, \dots, b_k | C | a_1, \dots, a_n) = \frac{1}{2^{n+k}} \begin{array}{c} a_1 \pi \\ \vdots \\ a_k \pi \\ a_{k+1} \pi \\ \vdots \\ a_n \pi \end{array} \begin{array}{c} \vdots \\ \vdots \\ \vdots \\ \vdots \end{array} \begin{array}{c} \vdots \\ \vdots \\ \vdots \\ \vdots \end{array} \begin{array}{c} b_1 \pi \\ \vdots \\ b_k \pi \\ \vdots \\ \vdots \end{array} \begin{array}{c} b_1 \pi \\ \vdots \\ b_k \pi \\ \vdots \\ \vdots \end{array} \begin{array}{c} \vdots \\ \vdots \\ \vdots \\ \vdots \end{array} \begin{array}{c} a_1 \pi \\ \vdots \\ a_k \pi \\ a_{k+1} \pi \\ \vdots \\ a_n \pi \end{array} \quad (2.47)$$

The doubling approach enables the result to be computed from just a single instance of strong simulation, rather than a number of instances exponential with

the number irrelevant qubits, m . This makes it the preferable choice when $m \approx n$ (i.e. $k \ll n$). However, this method also doubles the number of gates in the circuit, which for certain strong simulation approaches, results in squaring the computational complexity. Using ZX-calculus simplification, and given that the doubled circuit is mirrored, many of these gates may cancel, so that this method seldom doubles the number of gates in practice. Nevertheless, it does still increase the gate count, especially as $k \rightarrow n$. Consequently, the preferable choice of method depends on the size of the circuit and, more importantly, k .

Both of these methods utilise strong simulation to accomplish weak simulation. As such, devising more efficient strong simulation techniques in turn provides more efficient weak simulation, though indeed strong simulation is also a valuable task in its own right. While there are a number of methods for strongly simulating a circuit, as previously discussed, two are of particular interest to this thesis. These are tensor contraction and, more prominently, stabiliser decomposition.

2.3.3 Tensor Contraction

Tensor contraction (Ran et al., 2017) is a method — used in this context for strong simulation — based on the piecemeal merging of the tensors representing the gates in a quantum circuit. This idea is very similar to — and indeed a generalisation of — how individual constituent quantum gates were merged into larger compound gates in section 2.1.3.

Section 2.1.2 highlighted how an n -qubit quantum gate may be expressed as a uniform tensor of rank $2n$ and length 2. By extension, a quantum circuit is therefore a *tensor network*, with adjacent tensors (i.e. gates) sharing a number of indices equal to the number of wires directly connecting them.

Two vertices (tensors) in such a tensor network may be contracted into one by

summing over their common indices. For example:

$$\begin{array}{c} i \\ \text{---} \end{array} \bigcirc A_{ij} \begin{array}{c} j \\ \text{---} \end{array} \bigcirc B_{jk} \begin{array}{c} k \\ \text{---} \end{array} \quad \rightarrow \quad \begin{array}{c} i \\ \text{---} \end{array} \bigcirc C_{ik} \begin{array}{c} k \\ \text{---} \end{array} \quad (2.48)$$

shows the contraction of two tensors, A_{ij} and B_{jk} , into a new tensor C_{ik} . Meanwhile, i, j, k serve as edge labels and, as subscripts, denote which edges are connected to which tensors. The contraction itself is achieved by summing over the indices common to both tensors, $\{i, j\} \cap \{j, k\} = \{j\}$:

$$C_{ik} = \sum_j A_{ij} B_{jk} \quad (2.49)$$

This process may be iteratively repeated, contracting tensors pairwise until the entire tensor network is fully reduced to a single tensor. This is essentially the approach taken in equation 2.27. Applied to an n -qubit circuit, this results in contracting all gates into a single n -qubit compound gate, stored as a tensor of rank 2^{2n} tensor (or alternatively a $2^n \times 2^n$ matrix).

On the other hand, applied to a *scalar* ZX-diagram, tensor contraction ultimately results in a rank 0 tensor (no connected edges), which is simply a scalar. (Thus, given a circuit with closed inputs and outputs, this acts as a method for strong simulation.) Significantly, n -qubit circuits without open inputs and outputs (i.e. scalar diagrams) are generally contractable to its final scalar without requiring the full $O(2^{2n})$ memory overhead. The following example illustrates this point:

$$\begin{array}{c}
 \begin{array}{ccccc}
 A_i & \xrightarrow{i} & B_{ijk} & \xrightarrow{j} & C_j \\
 & & \downarrow k & & \\
 D_l & \xrightarrow{l} & E_{klm} & \xrightarrow{m} & F_m
 \end{array} \\
 = \\
 \begin{array}{ccccc}
 & & (AB)_{jk} & \xrightarrow{j} & C_j \\
 & & \downarrow k & & \\
 D_l & \xrightarrow{l} & E_{klm} & \xrightarrow{m} & F_m
 \end{array} \\
 = \\
 \begin{array}{ccccc}
 & & (ABC)_k & & \\
 & & \downarrow k & & \\
 D_l & \xrightarrow{l} & E_{klm} & \xrightarrow{m} & F_m
 \end{array} \\
 = \quad \dots \quad = \quad (ABCDEF)
 \end{array}
 \tag{2.50}$$

Each tensor is stored with $O(2^n)$ memory, where n is the number of indices it has (equivalent to the number of edges connected to it). In the above example, this means initially storing 6 tensors, A, B, \dots, F , with *at most* $O(2^3)$ memory for any given one (as the highest degree of any vertex here is 3).

The maximum memory used at any given time during the contraction is then given by the maximum degree⁹ vertex observed at any step during contraction. In this example, this never exceeds 3, ensuring a memory overhead of $O(2^3)$. However, had the corresponding circuit not been closed:

⁹The ‘degree’ of a vertex is simply the number of edges connected to it.

$$\begin{array}{c} i \\ \text{---} \end{array} \textcircled{B_{ijk}} \begin{array}{c} j \\ \text{---} \end{array} \\ \quad \quad \quad \downarrow k \\ \begin{array}{c} l \\ \text{---} \end{array} \textcircled{E_{klm}} \begin{array}{c} m \\ \text{---} \end{array} \quad = \quad \begin{array}{c} i \quad j \\ \text{---} \quad \text{---} \end{array} \textcircled{(BE)_{ijlm}} \begin{array}{c} l \quad m \\ \text{---} \quad \text{---} \end{array} \quad (2.51)$$

it would necessarily — regardless of contraction order — have resulted in a degree-4 vertex, given 4 open inputs/outputs, corresponding to $O(2^4)$.

Evidently, therefore, the space requirement when applied to scalar diagrams can, in general, be kept magnitudes below the theoretical maximum. It is important to contract the tensors in an appropriate *order* to ensure this memory requirement is minimised.

There remains ongoing research into tensor contraction to further improve its effectiveness. This includes parallel processing techniques (Abdelfattah et al., 2016), approximate methods (Gray & Chan, 2024), and heuristic optimisations of contraction paths (Orgler & Blacher, 2024). Recently, ZX-calculus has also been employed to aid in tensor contraction for classical simulation (Raj, 2022).

2.3.4 Stabiliser Decomposition

Another prominent approach to strong classical simulation is that of stabiliser decomposition, which involves translating circuits into sums of efficiently simulable stabiliser states.

Definition 13 (Stabiliser state). *A quantum state $|\psi\rangle$ is a stabiliser state if it can be generated by applying a Clifford circuit V to an initial computational basis*

$|0\rangle^{\otimes n}$ (Bravyi & Gosset, 2016):

$$|\psi\rangle = V |0\rangle^{\otimes n} \quad (2.52)$$

Definition 14 (Total stabiliser decomposition). *A ‘total’ stabiliser decomposition is a relation mapping a quantum state to a weighted sum of χ stabiliser states, where χ , known as its stabiliser rank, is the minimum number of such states required.*

In the context of ZX-calculus, a total stabiliser decomposition manifests as a relational equation mapping a $t > 0$ T-count ZX-diagram to a sum of χ Clifford ZX-diagrams.

Less strictly, a stabiliser decomposition may be defined as follows:

Definition 15 (Stabiliser decomposition). *A stabiliser decomposition is a relation mapping one quantum state to a weighted sum of η quantum states, each containing fewer non-Clifford gates.*

Expressed with ZX-diagrams, a stabiliser decomposition, by this definition, refers to a mapping between a $t > 0$ T-count ZX-diagram and a sum of η ZX-diagrams with T-counts $< t$.

With the use of *stabiliser decompositions*, it follows that:

Theorem 3. *Any quantum circuit can be expressed as a sum of stabiliser terms (Bravyi, Smith, & Smolin, 2016).*

Together with the Gottesman-Knill theorem (theorem 1), which states that any stabiliser circuit is efficiently simulable with classical hardware, one may conclude that any quantum circuit may be classically simulated by stabiliser decomposition (provided the number of stabiliser terms is not too large).

From the definitions of Z- and X- spiders in equation 2.32, it follows that:

$$\text{green circle} = \frac{1}{\sqrt{2}} \left(\text{red circle} + \text{red circle with } \pi \right) \quad (2.53)$$

Utilising the fusion and state copy rules, this may be generalised as such:

$$\begin{aligned} \text{green circle with } \alpha &= \text{green circle} - \text{green circle with } \alpha &= \frac{1}{\sqrt{2}} \left(\text{red circle with } \alpha + \text{red circle with } \pi \text{ and } \alpha \right) \\ &= \frac{1}{\sqrt{2}} \left(\text{red circle} + e^{i\alpha} \text{red circle with } \pi \right) \end{aligned} \quad (2.54)$$

A special case of this relation, with $\alpha = \frac{\pi}{4}$, provides a stabiliser decomposition for the so-called ‘magic state’ $|T\rangle \equiv \text{green circle with } \frac{\pi}{4}$ (Bravyi & Gosset, 2016):

$$\text{green circle with } \frac{\pi}{4} = \frac{1}{\sqrt{2}} \left(\text{red circle} + e^{i\frac{\pi}{4}} \text{red circle with } \pi \right) \quad (2.55)$$

More broadly, this stabiliser decomposition (or *T-decomposition*) may be applied to any T-like spider by first unfusing a magic $|T\rangle$ state:

$$\text{green oval with } (2n+1)\frac{\pi}{4} \text{ and } \vdots = \text{green circle with } \frac{\pi}{4} - \text{green oval with } n\frac{\pi}{2} \text{ and } \vdots \quad (2.56)$$

where $n \in \mathbb{Z}$.

Therefore, the T-decomposition of equation 2.55 may be used to exchange any T-like spider with a sum of two stabiliser terms. Applied to a ZX-diagram with T-count t , this hence produces two new (and mostly identical) ZX-diagrams, each of T-count $t - 1$. In turn, each of these two diagrams may have another T-spider removed with another application of equation 2.55. This doubles the total number of ZX-diagrams in the sum, but decrements the T-count of each, resulting in 4 ZX-diagrams of T-count $t - 2$. Iteratively repeating this process leads to 2^t Clifford

ZX-diagram terms.

In this way, any scalar Clifford+T ZX-diagram, such as arises from lemma 10, may be reduced to a sum of scalar *Clifford* ZX-diagrams (Backens, 2015; Coecke & Kissinger, 2018) which, per theorem 1, are efficiently simulable classically. Each may be reduced to scalar by the means of section 2.2.2, with their sum providing the scalar corresponding to the original scalar Clifford+T ZX-diagram. Hence:

Theorem 4. *Any Clifford+T quantum circuit may be strongly classically simulated via stabiliser decomposition, with a computational complexity that grows exponentially with its T-count.*

Figure 2.6 illustrates how recursively applying a stabiliser decomposition in this way reduces a non-stabiliser state into a sum of many stabiliser states¹⁰. While the time complexity here is $O(\eta^{t/\tau})$, computing this in a depth-first fashion avoids an exponential *space* complexity, needing instead only a memory overhead of $O(t/\tau)$. (Here, η is as defined in definition 15 and τ as defined in definition 16.) Consequently, time is the only expensive classical resource when strongly simulating Clifford+T ZX-diagrams with this approach.

Definition 16 (α efficiency). *A decomposition which reduces the T-count by τ and produces η terms may be recursively used to reduce a t T-count Clifford+T ZX-diagram to $2^{\alpha t}$ Clifford terms, where α quantifies the decomposition's efficiency*

¹⁰Instead of focusing on the stabiliser *rank*, χ , of a decomposition, an alternative metric one could consider is its stabiliser *extent*, ξ . This is defined as the sum of the absolute values of the coefficients of its stabiliser terms (Heimendahl, Montealegre-Mora, Vallentin, & Gross, 2021). By neglecting those terms whose coefficients (and likely contributions) are small, one may obtain an *approximate* solution more rapidly. This would amount to pruning less significant branches of the decomposition tree (figure 2.6). While there is some promising literature along these lines (Bravyi et al., 2019), such an approach is not considered any further in this thesis.

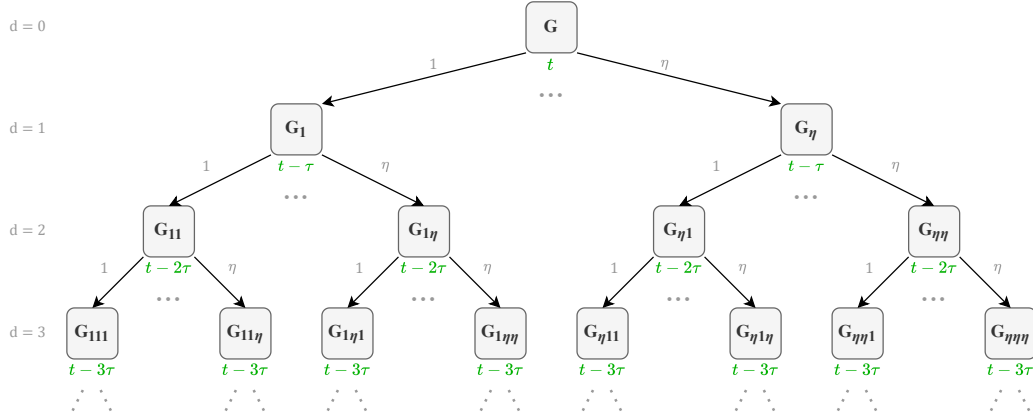


Figure 2.6: A decomposition tree, demonstrating how recursive applications of a stabiliser decomposition may translate a scalar Clifford+T ZX-diagram, G , of T-count t into a sum of many Clifford terms. Each node represents a scalar ZX-diagram with its T-count written beneath in green. Each application of the decomposition exchanges a ZX-diagram with η new ones of τ fewer T-spiders, such that at tree depth $d = \frac{t}{\tau}$ one is left with $\eta^{t/\tau}$ Clifford terms.

and may be defined as:

$$\alpha := \frac{\log_2 \eta}{\tau} \quad (2.57)$$

The particular decomposition presented in equation 2.55 exchanges $\tau = 1$ T-spiders for a sum of $\eta = 2$ stabiliser terms. As a result, using this to fully reduce a t T-count Clifford+T ZX-diagram would produce $\eta^{t/\tau} = 2^{\alpha t}$ Clifford terms, with an efficiency of $\alpha = 1$.

A more sophisticated stabiliser decomposition, introduced by Bravyi, Smith, and Smolin (Bravyi et al., 2016), and later applied to ZX-calculus in the work of Kissinger and van de Wetering (Kissinger & van de Wetering, 2022) is the ‘BSS’ decomposition (named for its authors):

$$\begin{aligned}
e^{i\pi/4} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} = 2e^{i\pi/4} \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ \circ \end{array} \\
-\frac{1+\sqrt{2}}{4} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} + \frac{1-\sqrt{2}}{4} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \\
-2\sqrt{2}i \begin{array}{c} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \\ \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ \circ \end{array} \end{array} -2i \begin{array}{c} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \\ \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ \circ \end{array} \end{array} \\
+8\sqrt{2}i \begin{array}{c} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \\ \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ \circ \end{array} \end{array} +8\sqrt{2}i \begin{array}{c} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \begin{array}{c} | \\ \circ \end{array} \\ \begin{array}{c} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ \circ \end{array} \end{array}
\end{aligned} \tag{2.58}$$

This decomposition exchanges $\tau = 6$ T-states for a sum of $\eta = 7$ stabiliser states and hence may reduce a t T-count ZX-diagram to a sum of $7^{t/6} \approx 2^{0.47t}$ Clifford terms¹¹. As such, it achieves an efficiency of $\alpha \approx 0.47$ — a considerable improvement upon the trivial decomposition of equation 2.55.

In the same paper, Kissinger and van de Wetering proposed using ZX-calculus to simplify not only the initial ZX-diagram but *every term* before they are decomposed. This can allow T-like spiders to fuse and cancel out (or disappear by other means, such as state copy), potentially reducing the T-count of each term along the decomposition tree and requiring a shallower depth to stabilise (i.e. reduce to Cliffords). This *inter-step simplification* means the number, N , of Clifford terms to which a Clifford+T ZX-diagram reduces is, in practice, fewer than that predicted by the decomposition's α efficiency. From this real measured N , and given an initial T-count t , one may quantify the practical (as opposed to theoretical) efficiency of the decomposition:

¹¹When the T-count is below 6, alternative decompositions, or simply direct calculation, may be used instead. So, to be precise, if t is not a multiple of 6 then this is an approximate result, or an *asymptotic* ($t \rightarrow \infty$) result.

Definition 17 (α_{eff} efficiency). *For a particular t T-count ZX-diagram, the measured **effective** α efficiency of a decomposition (or decomposition strategy) is given by:*

$$\alpha_{\text{eff}} := \frac{\log_2 N}{t} \quad (2.59)$$

where N is the number of Clifford terms to which it reduces.

As quantified in (Koch et al., 2023), and verified in experiments by the author of this thesis, the BSS decomposition with inter-step simplification achieves an average $\alpha_{\text{eff}} \approx 0.42$ on randomly generated Clifford+T+CCZ circuits (of non-trivial size). Notably, this is a significant reduction from the theoretical $\alpha \approx 0.47$ that is achieved *without* the use of inter-step simplification.

The subsequent work of Kissinger et al. (Kissinger et al., 2022) introduced the concept of *partial* stabiliser decompositions:

Definition 18 (Partial stabiliser decomposition). *A ‘partial’ stabiliser decomposition is a relation mapping one quantum state to a weighted sum of η quantum states, with at least one of which remaining non-Clifford.*

In particular, this work introduced a $|T\rangle^{\otimes 5}$ partial decomposition, as well as a family of total decompositions which apply to $|\text{cat}_n\rangle$ states (Qassim, Pashayan, & Gosset, 2021):

Definition 19. *A $|\text{cat}_n\rangle$ state is defined as:*

$$|\text{cat}_n\rangle := \frac{1}{\sqrt{2}} \left(\begin{array}{c} \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \\ \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \\ \vdots \\ \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \bigcirc \text{---} \end{array} \right) \quad (2.60)$$

These decompositions may be summarised as follows¹²:

¹²Using the π -commutation rule, these decompositions remain applicable with a π phase in

$$\begin{aligned}
& \text{Diagram 1} = \frac{e^{-i\pi/4}}{\sqrt{2}} \text{Diagram 2} + i \text{Diagram 3} \\
& \text{Diagram 4} = \frac{e^{-i\pi/4}}{\sqrt{2}} \text{Diagram 5} + i \text{Diagram 6} \\
& \text{Diagram 7} = \frac{1}{2} \text{Diagram 8} + \frac{ie^{i\pi/4}}{\sqrt{2}} \text{Diagram 9} - \frac{e^{i\pi/4}}{\sqrt{2}} \text{Diagram 10} \quad (2.61) \\
& \text{Diagram 11} = \frac{1}{2} \text{Diagram 12} + \frac{ie^{i\pi/4}}{\sqrt{2}} \text{Diagram 13} - \frac{e^{i\pi/4}}{\sqrt{2}} \text{Diagram 14}
\end{aligned}$$

Respectively, these achieve:

$$\begin{aligned}
\alpha_{|\text{cat}_3\rangle} &\approx 0.333, \\
\alpha_{|\text{cat}_4\rangle} &= 0.250, \\
\alpha_{|\text{cat}_5\rangle} &\approx 0.317, \\
\alpha_{|\text{cat}_6\rangle} &\approx 0.264.
\end{aligned} \quad (2.62)$$

Note that $|\text{cat}_1\rangle$ and $|\text{cat}_2\rangle$ states are reducible to Clifford by rewriting simplification alone, without the need for decomposition. $|\text{cat}_n\rangle$ states of $n > 6$, on the other hand, may be constructed from compositions of lower n cat states, though the same asymptotic efficiency in these cases may be achieved by simply using the following $|T\rangle^{\otimes 5}$ decomposition:

place of the 0 phase and, by spider (un)fusion, they remain applicable when any of its T-spiders are instead T-like.

$$\begin{array}{c} \pi/4 \\ \pi/4 \\ \pi/4 \\ \pi/4 \\ \pi/4 \end{array} = 2 \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} + 2\sqrt{2}ie^{i\pi/4} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} - 2\sqrt{2}e^{i\pi/4} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad (2.63)$$

In fact, this decomposition represents the state of the art for a $|T\rangle^n$ state, giving the lowest discovered asymptotic efficiency of $\alpha \approx 0.396$. While the decompositions of equation 2.61 offer lower α values than this, since they rely upon specific structures these are not universal. In other words, the above $|T\rangle^{\otimes 5}$ decomposition is applicable anywhere there are 5 or more T-like spiders, making it (at least asymptotically) universally applicable. The above cat decompositions, in contrast, are applicable only when specific structures (namely cat states) are present¹³. As such these cat decompositions, in general, are not alone sufficient to decompose any Clifford+T ZX-diagram, and will generally need to be used in conjunction with a universal decomposition such as that of equation 2.63.

With this in mind, Kissinger et al. (Kissinger et al., 2022) proposed a decomposition strategy whereby, following initial Clifford simplification, $|\text{cat}_n\rangle$ states of $n \leq 6$ are decomposed if present, with a preference order of ascending $\alpha_{|\text{cat}_n\rangle}$: $|\text{cat}_2\rangle$, $|\text{cat}_6\rangle$, $|\text{cat}_5\rangle$, and lastly $|\text{cat}_4\rangle$. If no such $|\text{cat}_n\rangle$ states are found then the universal $|T\rangle^{\otimes 5}$ decomposition of equation 2.63 is used instead.

As it turns out, $|\text{cat}_n\rangle$ states are ubiquitous among Clifford+T ZX-diagrams in reduced gadget form¹⁴. A fortunate consequence of this is that the ‘backup’ $|T\rangle^{\otimes 5}$

¹³In general, non-universal decompositions may incur an additional runtime overhead to search the ZX-diagram for applicable states. However, this is typically negligible compared to the runtime reduction offered by the decomposition, and certainly in the case of finding cat states this is trivial.

¹⁴As will be seen in chapter 5, it is easy to derive arbitrarily low α decompositions for highly specific, and conveniently designed, quantum states. The problem with this is that an extremely efficient decomposition for a quantum state that is so specific as to likely never to appear in practice is essentially useless. As such, it is important that decompositions act on reasonably general

decomposition is used rather rarely compared to the more efficient cat decompositions.

As this is a heuristic strategy, relying on several decompositions, it has no α . Nevertheless, its α_{eff} may be measured and quantified. Applied to randomly generated Clifford+T+CCZ circuits of non-trivial size, this decomposition strategy is found to give $\alpha_{\text{eff}} \approx 0.21$. This impressive result, being at one time the state of the art for a stabiliser decomposition approach to classical simulation, is commonly used as a baseline against which to compare new strategies.

More recent work by Koch et al. (Koch et al., 2023) improved upon this result, achieving $\alpha_{\text{eff}} \approx 0.19$ on the same class of circuits. This was accomplished by exploiting an alternative representation of the CCZ gate (Ng & Wang, 2018):

$$\begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} = 2 \begin{array}{c} \text{triangle} \\ \pi \\ \text{triangle} \end{array} \quad (2.64)$$

making use of the ‘*triangle*’ operation introduced in (Jeandel, Perdrix, & Vilmart, 2018):

$$\text{triangle} := \frac{1}{2} \begin{array}{c} \text{triangle} \\ \pi/4 \\ \text{triangle} \end{array} \quad (2.65)$$

together with a new set of decompositions optimised for this representation, with α efficiencies ranging from approximately 0.167 to 0.250:

structures that are (at least somewhat) likely to arise in practice.

$$\begin{aligned}
& \text{---} = \sqrt{2} \begin{array}{c} \circ \\ | \\ \circ \end{array} + 2 \begin{array}{c} \pi \\ | \\ \circ \end{array} \\
& \text{---} \text{---} = \frac{1}{\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} + \frac{1}{\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} + 4 \begin{array}{cc} \pi & \pi \\ | & | \\ \pi & \pi \end{array} \\
& \text{---} \text{---} \text{---} = \frac{1}{2\sqrt{2}} \begin{array}{cc} \circ & \circ \\ | & | \\ \circ & \circ \end{array} + \frac{1}{2\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} + \frac{1}{2\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} + \frac{1}{\sqrt{2}} \text{---} \text{---} + 8 \begin{array}{ccc} \pi & \pi & \pi \\ | & | & | \\ \pi & \pi & \pi \end{array} \\
& \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} = 3 \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} - \begin{array}{ccc} \pi & \pi & \pi \\ | & | & | \end{array} + \frac{3}{\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} - \frac{3}{2\sqrt{2}} \begin{array}{ccc} \pi & \pi & \pi \\ | & | & | \end{array} \\
& \begin{array}{c} \pm \frac{\pi}{2} \\ | \\ \circ \end{array} \begin{array}{c} \pm \frac{\pi}{2} \\ | \\ \circ \end{array} \begin{array}{c} \pm \frac{\pi}{2} \\ | \\ \circ \end{array} = \frac{1 \pm 3i}{2} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} + \frac{1 \mp i}{2} \begin{array}{ccc} \pi & \pi & \pi \\ | & | & | \end{array} - \frac{3-i}{2\sqrt{2}} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} \begin{array}{c} \circ \\ | \\ \circ \end{array} + \frac{1 \mp i}{2\sqrt{2}} \begin{array}{ccc} \pi & \pi & \pi \\ | & | & | \end{array} \\
& \begin{array}{ccc} \beta_1 & & \gamma_1 \\ \vdots & & \vdots \\ \beta_n & \alpha & \gamma_m \end{array} = \frac{1}{\sqrt{2}^n} \begin{array}{ccc} \beta_1 & & \gamma_1 \\ \vdots & & \vdots \\ \beta_n & & \gamma_m \end{array} + \frac{e^{i\alpha}}{\sqrt{2}^{n+m}} \begin{array}{ccc} \beta_1 + \pi & & \gamma_1 \\ \vdots & & \vdots \\ \beta_n + \pi & & \gamma_m \end{array}
\end{aligned} \tag{2.66}$$

where:

$$\text{---} := \begin{array}{c} \text{---} \end{array} \begin{array}{c} \pi \end{array} \text{---} = \begin{array}{c} \text{---} \end{array} \begin{array}{c} \pi \end{array} \begin{array}{c} \text{---} \end{array} \tag{2.67}$$

is a symmetric version of the triangle operation and known as a ‘*star edge*’.

This work nicely demonstrates how representing ZX-diagrams in a different manner can lead to more amiable decompositions.

As a significant portion of the related literature has focused on discovering new,

lower α , decompositions, a natural question to ask is: *how are new decompositions derived?* While there are a few approaches to this, ranging from trial and error to extending known decompositions for special cases, the most prevalent method is via *simulated annealing* (Bravyi et al., 2016; Van Laarhoven, Aarts, van Laarhoven, & Aarts, 1987; Rutenbar, 1989; Bertsimas & Tsitsiklis, 1993).

This method, broadly summarised in algorithm 1, involves initially selecting a set of stabiliser states — either by random selection or motivated by a heuristic — and assigning them random weights. These weights are then iteratively optimised through a type of random walk, biased towards improving fidelity (similarity between the weighted sum of stabiliser states and the target quantum state).

Simulated annealing was used extensively in (Laakkonen, 2022) to discover many competitive decompositions involving star edges (albeit using ‘*H-box*’ notation: $\text{---}\boxed{0}\text{---} \equiv \text{---}\text{---}\text{---}$). Among these was a $\text{---}\text{---}\text{---} \otimes^4$ decomposition into 5 stabiliser terms, achieving $\alpha \approx 0.145$, beating each of the decompositions of equation 2.66¹⁵. In the same work, the author experimented with using genetic algorithms as a means of finding new stabiliser decompositions, as well as a method based on composing states with known stabiliser decompositions in the hope of finding terms which cancel out.

Meanwhile, (Codsì, 2022) explored methods by which higher degree (i.e. greater n) $|\text{cat}_n\rangle$ states could be decomposed into (sometimes sums of) lower degree $|\text{cat}_n\rangle$ states. These methods involved elegant ‘*un-simplifications*’ (or *complications*) of $|\text{cat}_n\rangle$ states, together with a generalisation of equation 2.54:

$$\text{---}\text{---}\text{---} \otimes^4 = \left(\frac{1}{\sqrt{2}}\right)^n \left(\text{---}\text{---}\text{---} \otimes^4 + e^{i\alpha} \text{---}\text{---}\text{---} \otimes^4 \right) \quad (2.68)$$

¹⁵This has since been improved further, with (Vollmeier, 2025) achieving an $\alpha \approx 0.129$ decomposition for a $\text{---}\text{---}\text{---} \otimes^5$ state, using simulated annealing.

Algorithm 1 Simulated annealing method for finding stabiliser decompositions

- 1: **Input:** Target quantum state $|\psi\rangle$, set of candidate stabiliser states $\mathcal{S} = \{|s_1\rangle, \dots, |s_\chi\rangle\}$, initial temperature T_0 , cooling rate γ , and iteration count N
 - 2:
 - 3: Initialise random stabiliser state weights $\{w_1^{(0)}, \dots, w_\chi^{(0)}\}$, with $\sum_{i=1}^{\chi} |w_i|^2 \approx 1$
 - 4: Set initial temperature $T \leftarrow T_0$
 - 5:
 - 6: **for** $k = 1$ to N **do**
 - 7: Apply random small perturbations δw_i to current weights: $w'_i = w_i + \delta w_i$
 - 8: Normalise weights to satisfy $\sum_{i=1}^{\chi} |w'_i|^2 \approx 1$
 - 9:
 - 10: Compute fidelity: $F = |\langle \psi | \psi_{\text{approx}} \rangle|^2$, where $|\psi_{\text{approx}}\rangle = \sum_{i=1}^{\chi} w'_i |s_i\rangle$
 - 11: Compute change in fidelity: $\Delta F = F' - F$
 - 12:
 - 13: **if** $\Delta F > 0$ **then**
 - 14: Accept new weights: $w_i \leftarrow w'_i \ \forall i$
 - 15: **else**
 - 16: Accept new weights with probability $P = e^{\Delta F/T}$
 - 17: **end if**
 - 18:
 - 19: Update temperature: $T \leftarrow \gamma T$
 - 20: **end for**
 - 21:
 - 22: **Output:** Optimised weights $\{w_1, \dots, w_\chi\}$ for $|\psi\rangle \approx \sum_{i=1}^{\chi} w_i |s_i\rangle$ decomp.
-

where n is the number of outgoing edges. This simple decomposition is known as the ‘*vertex cutting*’ (or simply ‘*cutting*’) decomposition and is derived by unfusing a 1-degree spider from an n -degree spider (similar to equation 2.56) and applying equation 2.53 followed by state copy in each branch. Furthermore, as this decomposition allows an arbitrary number of edges, it is considered ‘*dynamic*’:

Definition 20 (Dynamic decomposition). *A stabiliser decomposition is said to be ‘dynamic’ if it allows for a variable number n of spiders and/or edges, such that its efficiency α is a function of n .*

A special case of the vertex cutting decomposition implements an *edge cut*:

$$\text{————} = \text{—} \circ \text{—} = \frac{1}{2} \left(\text{—} \bullet \bullet \text{—} + \text{—} \pi \pi \text{—} \right) \quad (2.69)$$

With this, two examples from the work of (Codsì, 2022) follow, demonstrating how larger $|\text{cat}_n\rangle$ states may be divided into smaller $|\text{cat}_n\rangle$ states:

$$\begin{aligned} \text{Diagram 1: } & \text{Cat state with } 2n \text{ legs (green spiders with } \pi/4) = \text{Cat state with } 2n+2 \text{ legs (two green spiders with } \pi/4) \\ & \approx \text{Cat state with } 2n \text{ legs and two green spiders with } \pi/4 + \text{Cat state with } 2n \text{ legs and two red spiders with } \pi \\ \text{Diagram 2: } & \text{Cat state with } 2n \text{ legs} = \text{Cat state with } 2n+2 \text{ legs (two green spiders with } \pi/4 \text{ and two yellow squares)} \\ & \approx \text{Cat state with } 2n \text{ legs and a chain of four spiders (green, red, green, red) with } \pi/4, \pi/2, \pi/4, \pi/2 \end{aligned} \quad (2.70)$$

The same approach was used to divide $|\text{star}_n\rangle$ states likewise:

Definition 21. A $|\text{star}_n\rangle$ state¹⁶ is defined as:

$$|\text{star}_n\rangle := \left. \begin{array}{c} \textcircled{\frac{\pi}{4}} \\ \textcircled{\frac{\pi}{4}} \\ \vdots \\ \textcircled{\frac{\pi}{4}} \end{array} \right\} n \quad (2.71)$$

In this way, (Codsi, 2022) found improved α decompositions for various high n $|\text{cat}_n\rangle$ and $|\text{star}_n\rangle$ states. Experimentally, this was found to produce an average $\alpha_{\text{eff}} \approx 0.32$ for random Clifford+T (without CCZ) ZX-diagrams, outperforming (Kissinger et al., 2022) which achieved $\alpha_{\text{eff}} \approx 0.35$ (Ahmad, 2024).

Overall, as this section highlights, there has been much research in recent years applying ZX-calculus to classical simulation. With its ease of interpretability and its powerful rewriting rules, ZX-calculus has proven a very natural and effective tool for this task. Many highly efficient (low α) stabiliser decompositions have been found (and continue to be found) with its aid, pushing the boundary of what is feasibly computable with classical hardware. In practical experiments, these improvements have significantly reduced the exponential growth rate of the runtime against the non-Clifford gate count when classically simulating quantum circuits. This has enabled ever larger (greater T-count) quantum circuits to be brought into the scope of computational feasibility for classical computers.

While the works discussed in this section have each demonstrated original techniques towards this end, there remains a common thread throughout. With a few exceptions, such as the original paper bringing these two areas of research together (Kissinger & van de Wetering, 2022) and related work on ZX-diagram partitioning (Codsi, 2022) (which is discussed further in section 2.4.3), the literature on applying ZX-calculus to classical simulation has focused on discovering new sta-

¹⁶Not to be confused with the unrelated star *edge* of equation 2.67.

biliser decompositions of ever lower α . This is, of course, a natural objective since lower α decompositions are expected to reduce the $O(2^{\alpha_{\text{eff}} t})$ computational cost of simulating t T-count circuits.

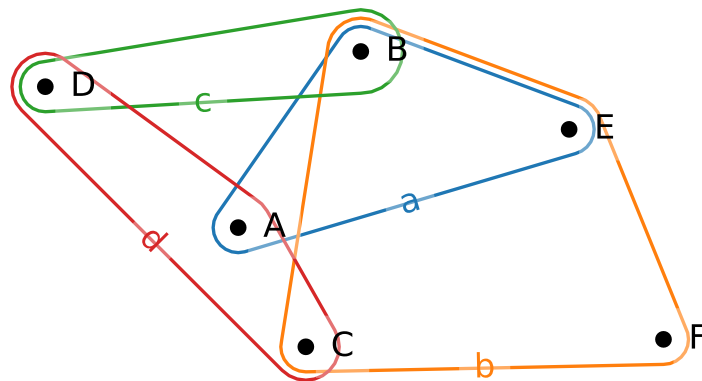
However, in a break from tradition, this thesis (with the exception of chapter 6, depending on interpretation) avoids searching for lower α decompositions (or new methods of finding them). Instead, the novel work introduced in the subsequent chapters explores altogether new ideas for how classical simulation may be optimised using ZX-calculus.

2.4 Hypergraph Partitioning

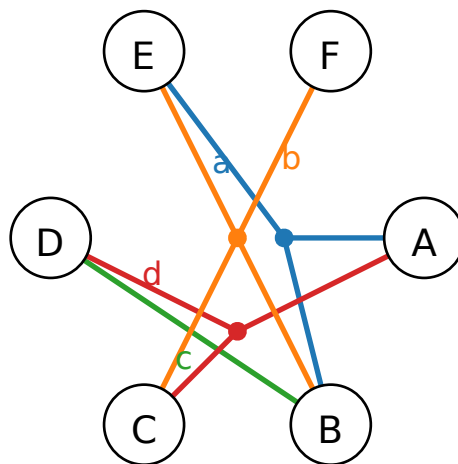
Throughout this thesis, quantum circuits are typically expressed as ZX-diagrams, as detailed in section 2.2. Notably, a ZX-diagram is a type of graph and as such it may take advantage of known techniques from graph theory literature. In particular, the well-established concept of hypergraph partitioning will play a crucial role in chapter 4.

2.4.1 Weighted Hypergraphs

A *hypergraph* is a generalisation of a graph, with vertices connected by *hyperedges* rather than conventional edges. The difference is that hyperedges are able to connect any arbitrary number of vertices, whereas conventional edges may only connect two vertices (Bretto, 2013). Most commonly, hypergraphs are visualised as a set of vertices, A, B, \dots , with hyperedges, a, b, \dots , represented by nets encompassing some subset of these vertices, like so:



However, for reasons that will hopefully become apparent in chapter 4, throughout this thesis hypergraphs will instead be portrayed as vertices connected by n -ended hyperedges, such that the above example may be visualised like so:



Furthermore, with the inclusion of *weights* on each of the vertices and hyperedges, one may define a *weighted hypergraph*:

Definition 22. Formally, a weighted hypergraph $H = (V, E, c, \omega)$ consists of:

- a set of vertices $V = \{v_1, v_2, \dots, v_n\}$,
- a set of hyperedges $E = \{e_1, e_2, \dots, e_m\}$, where $e_i \subseteq V$ for each $i \in \{1, 2, \dots, m\}$,

- a function $c : V \rightarrow \mathbb{R}^+$ assigning a weight to each vertex, and
- a function $\omega : E \rightarrow \mathbb{R}^+$ assigning a weight to each hyperedge.

(Schlag, 2020)

2.4.2 Minimum Balanced k -Cut

In graph theory, there exist a number of variations of the *graph partitioning problem*, where the aim is to partition a graph into k disjointed subgraphs while minimising (or maximising) some objective function. In each case, a generalisation for hypergraphs may typically also be defined. One such variation of interest to this thesis is the *minimum balanced k -cut problem*, which may be defined for hypergraphs as follows:

Definition 23. *Given a hypergraph $H = (V, E, c, \omega)$, the minimum balanced k -cut problem aims to partition V into $k \in \mathbb{Z}^+$ balanced disjointed subgraphs $\{V_1, V_2, \dots, V_k\}$, such that:*

- *the total summed weight of ‘cut’ hyperedges (i.e. those connecting vertices between disjointed subgraphs) is minimised:*

$$\text{Minimise } \sum_{e \in C} \omega(e) \quad (2.72)$$

where $C \subseteq E$ is the set of hyperedges that connect vertices across two or more subgraphs, and

- *the partitioned subgraphs are ‘balanced’, meaning each subgraph has an approximately equal sum of vertex weights, up to some allowed imbalance coefficient $\epsilon \geq 0$:*

$$\sum_{v \in V_i} c(v) \leq (1 + \epsilon) \cdot \frac{1}{k} \sum_{v \in V} c(v) \quad \forall i. \quad (2.73)$$

(Schlag *et al.*, 2022)

Solving this problem exactly, or even approximately, is NP-hard (Bui & Jones, 1992; Lengauer, 2012). Nevertheless, good solutions can generally be found within reasonable times via heuristic methods, such as that offered by the *KaHyPar* package (Schlag, 2020).

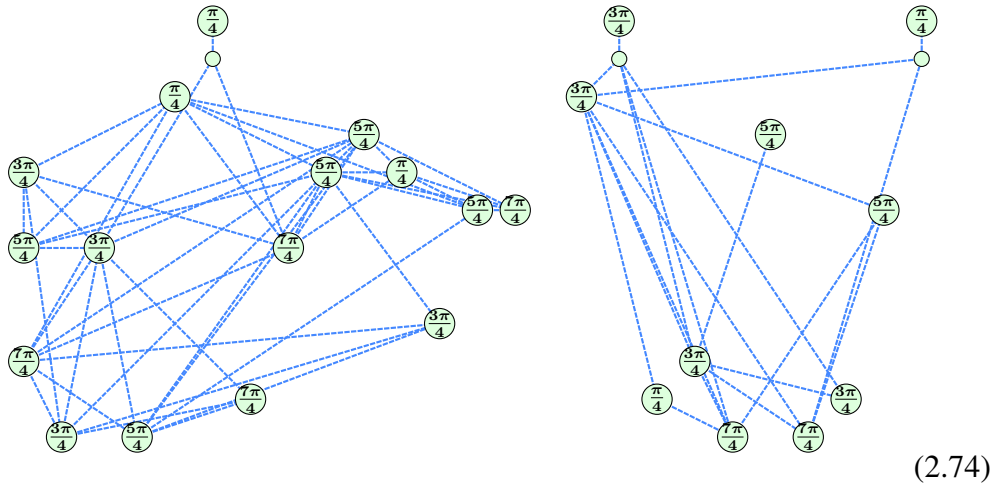
KaHyPar’s method involves an initial *coarsening* phase, where heuristics are used to group together local vertex clusters based on their connectivity and hyperedge weights. This essentially lowers the resolution of the hypergraph and allows it to be simplified into a smaller approximate counterpart. This smaller ‘coarsened’ hypergraph can then be efficiently k -partitioned, before a phase of *uncoarsening* and local refinement (Ravikumār, 1996) is employed to progressively account for the lost details and refine the solution. This makes use of modified versions of traditional graph partitioning algorithms such as those of (Kernighan & Lin, 1970) and (Fiduccia & Mattheyses, 1988).

The intricacies involved are fairly complex, though for the scope of this thesis KaHyPar can be treated as a black box function for finding good and quick heuristic solutions to this partitioning problem, given a hypergraph H , a partition count k , and a balance tolerance ϵ . This will be used notably in chapter 4 for partitioning ZX-diagrams.

2.4.3 Partitioning ZX-Diagrams

Previous sections of this chapter demonstrated how a Clifford+T ZX-diagram may be simplified via the rewriting rules to reduce its T-count. It was further shown

that, given a reduced T-count t , the ZX-diagram may be strongly classically simulated via decomposition into $2^{\alpha t}$ stabiliser terms. In fact, it is possible, after initial Clifford simplification, for the ZX-diagram to reduce to two or more separated subgraphs, such as follows:



In such cases, each subgraph may be independently reduced to scalar, with the overall scalar result given by their product:

Lemma 4. *A t T-count Clifford+T ZX-diagram composed of $k \geq 2$ disjoint sub-diagrams:*

$$\boxed{G} \equiv \boxed{G_1} \boxed{G_2} \cdots \boxed{G_k} \quad (2.75)$$

may be more efficiently reduced to scalar by reducing each of its k sub-diagrams independently and taking the product of the resulting scalars. If each sub-diagram, i , contains $t_i \leq t$ T-spiders, such that $\sum_{i=1}^k t_i = t$, then the overall ZX-diagram

may be strongly classically simulated via

$$\sum_{i=1}^k 2^{\alpha t_i} \leq 2^{\alpha t} \quad (2.76)$$

stabiliser terms (Codsì, 2022).

Where applicable, this is liable to offer an exponential speedup versus naïvely processing the ZX-diagram as a whole. For instance, if a ZX-diagram, G , may be expressed by 2 disconnected subgraphs, G_A and G_B (of T-counts t_A and t_B respectively), the best case scenario would divide the overall T-count, t , evenly between the two subgraphs: $t_A = t_B = \frac{t}{2}$. This would reduce the number of stabiliser terms to compute from $2^{\alpha t}$ down to $2 \cdot 2^{\alpha t/2}$ (Codsì, 2022).

Naturally occurring partitions (via the rewriting rules) are not uncommon, though seldom exist beyond very low k and $\max(t_i) \approx t$. Such cases tend to offer small, but not necessarily insignificant, runtime reductions. However, more ideal partitions, such as the ‘best case’ bipartition described above, seldom arise naturally.

Nevertheless, such partitions can be induced by applying some instances of a locally partitioning decomposition. In particular, the *edge cutting* decomposition introduced in equation 2.68 allows any edge to be removed from a ZX-diagram at the cost of doubling the number of terms. Consequently, any ZX-diagram may be partitioned into arbitrarily many subgraphs, given enough cuts:

Lemma 5. *A ZX-diagram may be partitioned into $k \geq 2$ disjointed parts at the cost of 2^c terms, given c cuts:*

$$(2.77)$$

Thus, the ZX-diagram may be strongly classically simulated via:

(2.78)

stabiliser terms, given subgraph T-counts $\{t_1, t_2, \dots, t_k\}$.

In fact, it is more efficient to apply *vertex cuts* (equation 2.68) rather than edge cuts, as the former can be interpreted as a generalisation of the latter (having the effect of cutting all of its edges individually). The goal, therefore, is to find a minimal set of spider cuts that partitions a given ZX-diagram into $k \geq 2$ disjointed subgraphs of roughly equal T-counts:

Lemma 6. *A ZX-diagram of T-count t is optimally partitioned into $k \in \mathbb{Z}_{\geq 2}$ disjoint subgraphs of T-counts $\{t_1, t_2, \dots, t_k\}$ when:*

- the number of vertex cuts c is minimised:

(2.79)

and

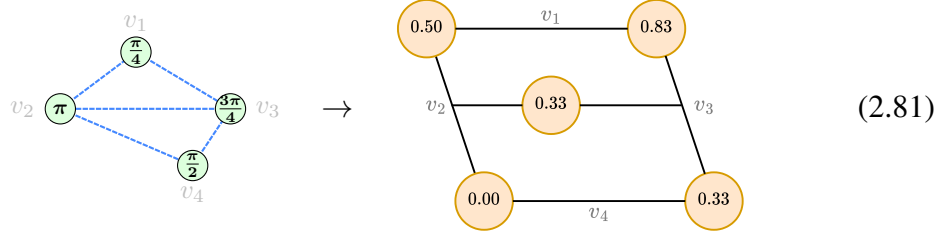
- *each subgraph contains an approximately equal number of T-spiders, up to some imbalance $\epsilon \in \mathbb{R}_{>0}$:*

(2.80)

In graph theory, this is known as a *vertex separator problem* (Rendl & Sotirov, 2018; Althoby, Biha, & Sesboüé, 2020). However, it may also be mapped to a *hypergraph partitioning problem* (Gottesbüren, Heuer, Maas, Sanders, & Schlag, 2024), in which hyperedges are cut rather than vertices, as this has more extensive literature and tools available. Specifically, this problem may be mapped to the *minimum balanced k -cut problem* defined in section 2.4.2.

Before mapping to this problem, relevant vertex and edge weights need to be included. As such, each spider receives a weight of 1 as — ignoring any Clifford simplification that may be facilitated — the cost of a vertex cut is essentially the same regardless of the vertex being cut. Meanwhile, for each T-spider an edge weight of $\frac{1}{n}$ is added to each of its n edges. This serves to ‘spread’ the weight due to the T-spider into its edges (which will soon be translated into vertices).

Thereafter, the ZX-diagram must be translated into a hypergraph. This can be achieved simply by exchanging each edge for a vertex and each spider for a hyperedge like so:



Here, each vertex in the ZX-diagram, and its corresponding hyperedge in the hypergraph, is labelled in grey, and the vertex weights in the hypergraph are shown in black to two decimal places.

With the ZX-diagram translated to a weighted hypergraph, it may be efficiently k -partitioned using the KaHyPar software described in section 2.4.2.

Such a method was employed in (Codsì, 2022) to bipartition ZX-diagrams with the aim of speeding up strong classical simulation. Therein, it was found that for random Clifford+T circuits, efficient partitions were rare due to their dense edge connectivity.

In (Codsì, 2022), this method focused exclusively on *bipartitions* (i.e. k -partitions where $k = 2$). This is presumably because, with the method as presented, balanced k -partitions beyond $k \gtrsim 3$ are generally unhelpful due to the number of cuts involved outweighing the effective T-count reduction. In other words, in k -partitioning a given ZX-diagram up to some imbalance ϵ , the number of cuts required typically (but not necessarily) grows at least proportionally to k .

Consequently, larger k tends to result in a reduced $\sum_{i=1}^k 2^{\alpha t_i}$ factor (see lemma 5) but an *increased* 2^c factor. This means, generally speaking, for a given ZX-diagram there is some k beyond which k -partitioning actually performs worse than direct stabiliser decomposition. Anecdotally, on typical Clifford+T ZX-diagrams this crossover point tends to be very low at around $k \approx 3$.

2.5 GPU Parallelism

Since the late 1990s and early 2000s, commercial computers have come equipped with a *Graphics Processing Unit* (GPU)¹⁷ — a specialised circuit designed, as its name suggests, for efficiently computing graphics, particularly for gaming (Peddie, 2023). Unlike CPUs, which are designed for fast and general purpose sequential processing, GPUs are very effective at computing very simple procedures on elements of a large dataset in parallel.

¹⁷To be precise, there is a distinction between the ‘GPU’, being solely the processor component, and the ‘graphics card’, being the whole hardware unit including the GPU and its memory and interfaces. However, this thesis is not concerned with such pedantry and, consistent with colloquial use, will use the terms interchangeably.

Its potential for applications beyond graphics was soon recognised, giving rise to *general purpose GPU* (GPGPU) programming. Since the late 2000s GPUs have been utilised for the likes of machine learning (Raina, Madhavan, Ng, et al., 2009; Lopes & Ribeiro, 2011), physics simulations (Harada, 2007; Nylons, Harris, & Prins, 2007; Crane, Llamas, & Tariq, 2007; Alerstam, Svensson, & Andersson-Engels, 2008), bioinformatics (Li et al., 2012; Nobile, Cazzaniga, Tangherloni, & Besozzi, 2017; Hasan, Chatterjee, Radhakrishnan, & Antonio, 2014), and more (Whalen, 2005; Criminisi, Sharp, Rother, & Pérez, 2010; Garcia, Debreuve, & Barlaud, 2008).

This section aims to cover the relevant background of GPGPU programming and the limitations and considerations involved when taking advantage of GPU parallelism. As this is a very in-depth and nuanced topic with many intricacies, its coverage in this chapter will be focused to a scope directly relevant to the later chapters, with many details omitted and explanations oversimplified.

2.5.1 Parallel Processing

It is firstly important to clarify what is meant by parallelism and its distinction from concurrency:

Definition 24. *Parallelism refers to events occurring simultaneously. In computing, this requires multiple independent processors or cores executing different tasks at the same time.*

Definition 25. *Concurrency refers to the concept of managing multiple tasks at once, though not necessarily at the same time. This is often handled through interleaving the tasks and can often give the illusion of parallelism. A single independent processor may process multiple tasks concurrently, though only one is ever being processed at any given moment.*

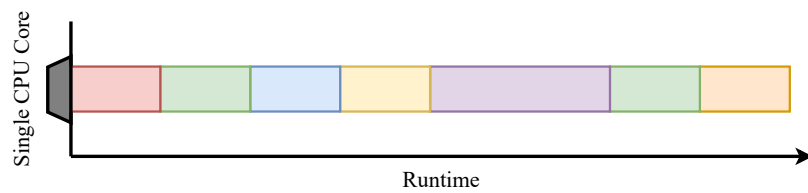
The following example of concurrency emphasises this distinction:

Example 1. *The 1983 computer game, Manic Miner, famously featured background music alongside event-based sound effects — a feat previously considered impossible on the limited hardware of the ZX-Spectrum home computer, which contained only a single sound channel (McAlpine, 2015). This was accomplished with a technique known as ‘arpeggiated multiplexing’, whereby the sound channel rapidly alternated between the background music and sound effects, producing a cohesive audio experience and giving the illusion of both being played simultaneously. In this instance, the music and effects were played concurrently but not in parallel.*

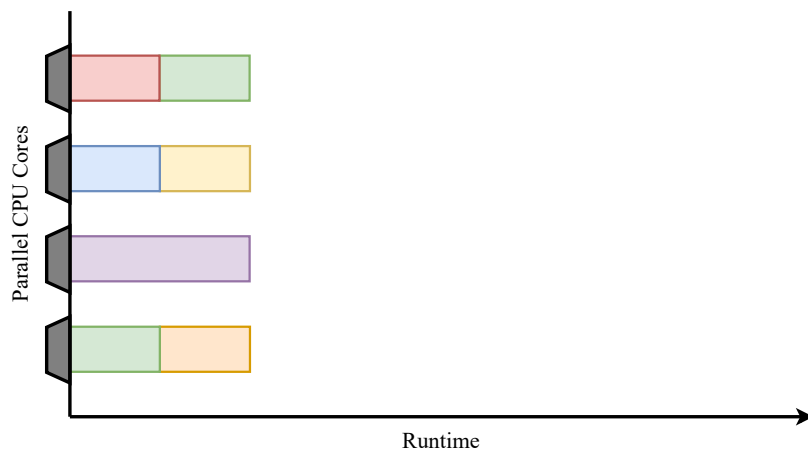
The simplest type of parallel processing is *CPU* parallelism. Since IBM released the first commercial multicore processor in 2001 (Stallings, 2010), CPUs with increasing core counts have become commonplace. Modern commercial CPUs typically feature 4, 8, or even 16 cores, while high-performance workstations and server CPUs may have significantly more. In simple terms, an n -core CPU behaves much like n individual processors, being able to execute up to n independent tasks in parallel (albeit while sharing memory and cache bandwidth) (Sanders & Kandrot, 2010).

Figure 2.7 highlights this point, showing how 7 independent and colour-coded tasks may be processed either sequentially with a single CPU core or in parallel with 4 CPU cores. Aside from rather trivial considerations, such as distributing the tasks (by total runtime) as evenly as possible across the cores, there is very little effort or difficulty involved in parallelising tasks in this way. The only notable requirement is that tasks executed in parallel must be independent of one another (that is, the input of one cannot depend upon the output of another).

CPU parallelism, therefore, can be a very simple and easy way to improve the



(a) Sequential processing of 7 independent tasks.



(b) Parallel processing of 7 independent tasks.

Figure 2.7: A simple illustration of how 7 independent tasks may be processed on (a) a single CPU core versus (b) 4 parallel CPU cores.

overall runtime for computing a set of tasks by a small factor (no greater than the number of cores). However, much more drastic results can be achieved with the use of GPU parallelism, but this is much less universally applicable, being suitable to a much narrower set of scenarios.

Parallelising tasks for a GPU is much more intricate and nuanced than doing so for a CPU, and it is only viable when specific conditions are met. The most important of these is outlined in remark 2. Essentially, GPUs are designed to compute simple instructions to a large set of data simultaneously, which is in contrast to CPU cores which act rather as independent processors able to compute independent procedures in parallel. Figure 2.8 demonstrates a very simple and

trivial example of SIMD processing.

Remark 2. GPU architecture is based on a ‘Single Instruction Multiple Data’ (SIMD) model of computation (Hennessy & Patterson, 2011). This means each parallel thread must execute the same procedure, applied to a different unit of data, with each instruction executed in lockstep¹⁸.

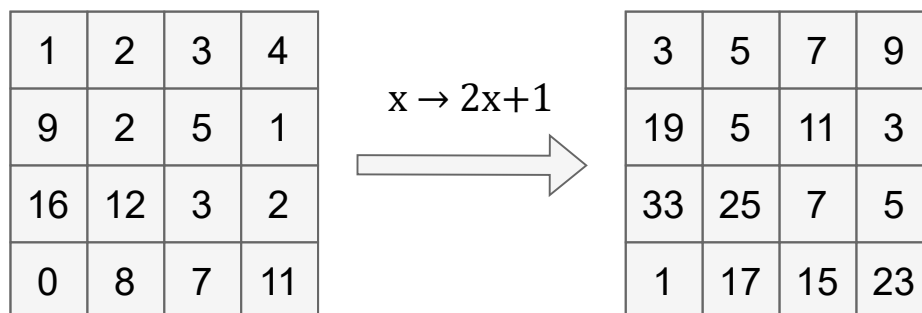


Figure 2.8: A simple example of SIMD processing, where one function is applied to many data simultaneously.

Figure 2.9 illustrates the processing pattern of tasks executed on a GPU. Comparing this to figure 2.7, one may make a few observations:

1. There are many more parallel ‘threads’ available on a GPU as compared to parallel cores on a CPU. Indeed, typical commercial GPUs are equipped with many thousands of threads capable of executing in parallel.
2. Each thread must execute the same function (known as a ‘kernel’) at the same time, albeit applied to different data. This is a consequence of the GPU’s SIMD model.
3. Each process on a GPU is significantly slower than its equivalent on a CPU. This is because GPU threads are individually much less powerful than CPU cores, having a greatly reduced instruction set.

¹⁸This is a slight oversimplification, as different ‘warps’ of 32 threads may execute out of sync.

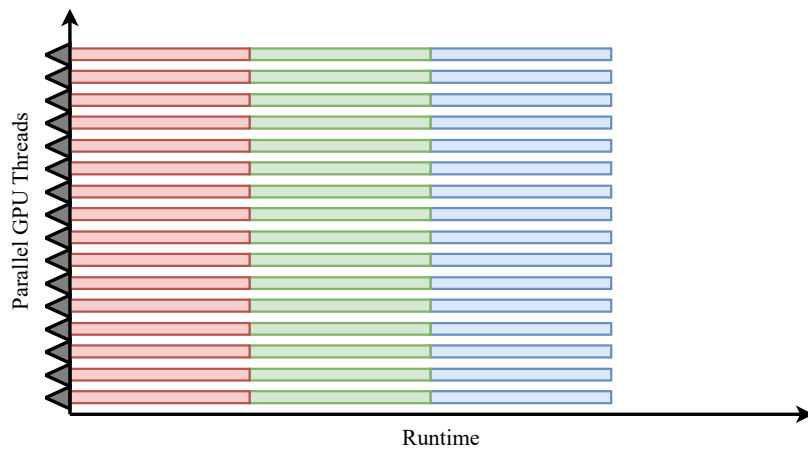


Figure 2.9: A simple illustration of how three independent (and colour coded) tasks are executed across many threads simultaneously on a GPU.

From each of these three points arises a notable implication:

1. To make full use of the GPU, it should be run at full capacity, meaning it is best used when given a very large amount of data to process.
2. Due to the GPU's SIMD execution model, all threads (within a 'warp' of 32) must follow the same instruction path at the same time. When a branch occurs (such as via an *If* statement), some threads may take one path while others take another. However, since all threads must execute both paths (with inactive threads idling during its 'false' branch), this leads to inefficiencies known as *warp divergence*. Similarly, iteration loops (such as *For* loops) should have a consistent iteration count across all threads to avoid uneven execution times and idle threads. Therefore, to maximize efficiency, one should ensure uniform workloads across all threads and avoid branching code.
3. As GPU cores are much less powerful than CPU cores, GPU kernels should ideally rely only upon simple arithmetic, with data recorded in very prim-

itive data structures. Higher-level data structures and operations are not generally viable as these would slow down operations drastically.

These restrictions (among others) render GPU parallelism much less universally applicable and much less trivial to implement, as compared to CPU parallelism. However, where it is applicable (or can be made so) it is liable to offer much more drastic speedups in overall runtime, as will be seen in chapter 3.

2.5.2 GPU Architecture

GPUs execute a special type of function called a ‘*kernel*’, with many parallel threads executing the function simultaneously (Sanders & Kandrot, 2010). Each individual thread (and each block of threads) has a unique index number which may be read within the kernel. This allows many threads to execute the same function, but applied to different data. Algorithm 2 provides pseudocode for a very simple example which, used as here, turns an array of zeroes into an array of ones.

Algorithm 2 A very simple example of a GPU kernel.

```

1: function INCREMENT(data)  $\ll n \gg$                                  $\triangleright$  Define GPU kernel
2:    $i \leftarrow \text{GETTHREADINDX}()$                                  $\triangleright$  Get unique thread index
3:    $data[i] = data[i] + 1$ 
4: end function
5:
6:  $nums[1024] \leftarrow 0$                                  $\triangleright$  Define length-1024 array of zeroes
7:  $data \leftarrow \text{MEMTOGPU}(nums)$ 
8: INCREMENT(data)  $\ll 1024 \gg$                                  $\triangleright$  Execute the kernel with 1024 threads
9: SYNCHRONISETHREADS()
10:  $nums \leftarrow \text{MEMFROMGPU}(data)$ 

```

Here, INCREMENT is a kernel which takes as input an array, *data*, and executes on *n* threads, and GETTHREADINDX returns the unique thread index as a number between 1 and *n*. This kernel, therefore, increments the first *n* elements of the

array, *data*, in parallel. All the code outside of the kernel is executed on the CPU, while the kernel itself (when called) is executed on the GPU.

In this example, *nums* is defined as a length-1024 array of zeroes, which line 7 then sends from the CPU to the GPU. The copy of the data on the GPU is then labelled *data*. Note that this `MEMTOGPU` function completes in full before proceeding to the next line (though an alternative `ASYNCMEMTOGPU` may be used to asynchronously send the data to the GPU, meaning the next line will execute right away, while this data transfer is still underway, but this should be used with caution).

Line 8 then executes the kernel with 1024 threads, and passing in *data* (which is already stored in GPU memory). As this kernel is executed on the GPU, the CPU is immediately free after making this call, meaning it may continue onto its subsequent lines while the GPU processes this request. In this example, it is essential to ensure the kernel completes for all threads before running the next line and so `SYNCHRONISETHREADS()` forces the CPU to wait for the GPU to finish processing the kernel for all threads before continuing. This ensures that when line 10 returns *data* from the GPU to the CPU, and pastes the result into *nums*, that every element has been updated.

GPUs are highly specialised and process data in a hierarchical structure to very efficiently optimise for its SIMD processing model. As such, there are several tiers of abstraction at which threads are processed, as outlined ahead (NVIDIA, 2020):

- A **thread** is the smallest unit of execution, being a single instance running a kernel with a unique thread identifier. It has its own registers and local memory.
- A **warp** is a group of 32 threads which execute in lockstep, executing kernel

instructions in perfect unison. Any warp divergence will result in the whole warp being slowed to the rate of its slowest member and should be avoided.

- A **block** is a group of warps, typically containing a collective 128 to 1024 threads, with shared memory and cache and some degree of synchronisation.
- A **grid** contains the entire set of blocks that execute a kernel. Together with global (and constant) memory, the grid essentially encompasses the entire scope of the GPU.

Each of the above refers to an abstract (logical) concept. The physical hardware components that process these are as follows (NVIDIA, 2020):

- A **GPU core** (often referred to as a *CUDA core* for Nvidia brand hardware) is the smallest processing unit, which executes a single thread at a time¹⁹. They execute in parallel and each acts as an individual processor, performing arithmetic and logical operations to the data assigned to it.
- A **streaming multiprocessor (SM)** is broader processing unit, managing many cores plus data transfer, memory, caching, and scheduling of threads and warps.
- A **graphics processing unit (GPU)** is the entire hardware unit encompassing the above.

One important takeaway from these points is the distinction between a thread and a core. A thread is a single unit of execution of a kernel, while a core is physical hardware which processes a thread. The former has no limit, while the latter is finite. One may execute a kernel to process, for example, 100,000 threads. If the

¹⁹In fact, with appropriate scheduling to minimise idle thread time (such as when waiting for data access), each core will process many threads concurrently, **but not simultaneously**. For simplicity, however, this point may be overlooked.

number of cores available were 1,000 then (to greatly oversimplify) these threads would be processed in 100 parallel batches of 1,000. Hence, **not all threads execute in parallel**, though to avoid frequently clarifying this point it is common, when speaking abstractly, to colloquially describe them as though they do (for instance, “*this kernel processes these n threads in parallel.*”).

2.5.3 GPU Memory Structure

A GPU contains various types of memory and cache, accessible within different scopes and generally serving different purposes. The most significant of these, together with their primary uses, are as follows (NVIDIA, 2020; Farber, 2011):

- **Global memory** is used for storing large datasets that are accessed by all threads across all blocks. This will typically include the main dataset to be processed on the GPU, with portions of it often later sent into shared memory for faster access by the threads.
- **Constant memory** is designed for read-only data that is uniform across all threads, such as constants and parameters that remain unchanged during runtime.
- **Shared memory** is common to threads within the same block and benefits from efficient data sharing and synchronisation between them. It often acts as a fast cache to hold portions of global memory that threads within a block need repeated access to.
- **Registers** are used to store temporary variables and intermediate results local to individual threads. This includes, for example, loop counters and any primitive variables, such as integers or floats, that are declared during runtime. Registers are the only non-persistent memory type among these, meaning its data only exists during the runtime of a particular thread.

- **Local memory** is private to each individual thread and is mostly used to store runtime variables that are too big to be stored in registers. This includes most non-primitive data structures, such as arrays and structs. It is also used as an overflow for primitive variables if the thread runs out of register space. Physically, each thread's local memory is a partitioned segment of global memory, but being allocated to a specific thread means access patterns are localised and contention is minimised. This, in practice, renders local memory quicker than global memory.

Table 2.1 provides a broad overview of the speeds and scales of each of these memory types, together with their access scope (NVIDIA, 2020; Farber, 2011). In addition to the *speed*, which describes how quickly data can be read and written, the *latency* (time delay between requesting data from memory and receiving it) and *bandwidth* (rate, in GB/s, at which data can be transferred to and from memory) of each memory type are also important factors to consider, though these are neglected here for simplicity.

Name	Access	Speed	Size
Global Memory	All threads	Slow	Large (~4–24GB)
Constant Memory	All threads (read-only)	Fast	Small (~64KB)
Shared Memory	Threads within the same block	Very Fast	Small (~48KB per block)
Local Memory	Per thread	Medium	Small (~8–16KB per thread)
Registers	Per thread	Fastest	Tiny (~32 bytes per thread)

Table 2.1: Overview of the main GPU memory types (Lai et al., 2019).

Figure 2.10 provides a visual overview of this memory structure (Lai et al., 2019).

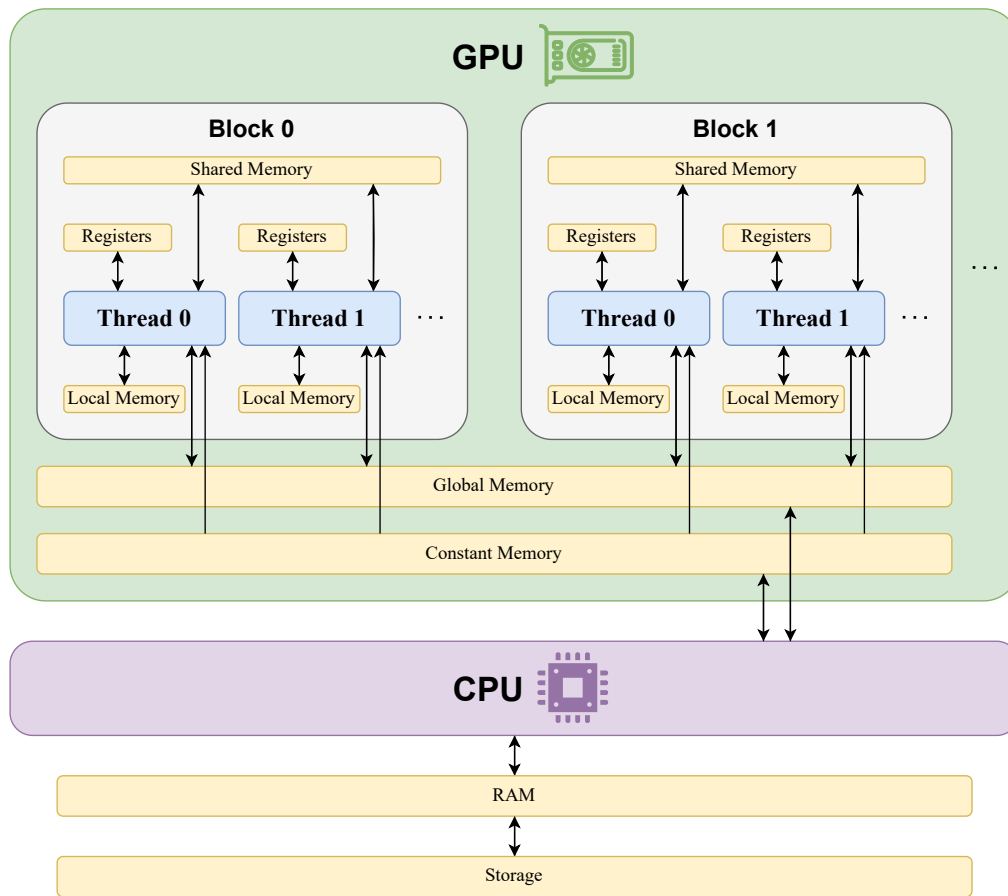


Figure 2.10: The memory structure of a GPU (Lai et al., 2019).

Note that the initial data is sent from the CPU to the global (and/or constant) memory of the GPU, and, after processing, the result will be read back to the CPU from global memory.

2.5.4 Data Coalescing

To ensure high performance computing on a GPU, it is essential to take extra care to optimise the data access patterns. As reading data from global memory is very slow compared to shared memory and registers, the number of such transactions should be minimised. This can often be achieved by organising the data in such a

way as to ensure the data requested by individual threads within a block are stored among consecutive memory locations, such that they may be transferred in fewer bulk transactions rather than many individual ones. This is known as *coalescing* the data (Farber, 2011). Example 2 demonstrates this concept.

Example 2. *As an example prescient of chapter 3, consider the 5×6 matrix presented in table 2.2, where $a_{ij} \in \mathbb{Z} \forall i, j$, and suppose one desired to sum the elements of each row. In other words, for each $i \in \{1, 2, \dots, 5\}$ compute $a_i \in \mathbb{Z}$ where:*

$$a_i = \sum_{j=1}^6 a_{ij} \quad (2.82)$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}

Table 2.2: A simple example of two-dimensional data, where $a_{ij} \in \mathbb{Z} \forall i, j$.

While it is convenient to interpret such data as two-dimensional, in memory it is necessarily stored linearly. Ordinarily, this would be stored in row-major order, like so:

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{31}	a_{32}	\dots
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	---------

With typical sequential processing (such as by a single core CPU), each row would be computed, via equation 2.82, one after the other. This means initialising $a_1 = 0$, and then adding a_{11} , and then a_{12} , and so on to a_{16} , before next moving onto the next line to begin computing a_2 . With this in mind, it is clear that row-major ordering provides the most efficient data access pattern, with subsequently accessed elements being physically stored in adjacent memory addresses.

Now consider instead processing this data on a GPU, with each row being processed in parallel on its own thread. In this scenario, after initialising $a_i = 0 \forall i$, at the first time step, each thread i would, in parallel, compute $a_i = a_i + a_{i1}$. This means, at the first time step, every a_{i1} would be needed at the same time, as is highlighted in blue in table 2.2. Evidently, linearising this data with row-major ordering means that each of these elements are now far apart in memory.

Instead, to ensure the parts of data that are needed at the same time are physically near each other and hence can be accessed in fewer transactions and more rapidly, it would be more efficient to store this data in column-major order, like so:

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	a_{12}	a_{22}	a_{32}	a_{42}	a_{52}	a_{13}	a_{23}	a_{33}	a_{43}	\dots
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	---------

Essentially, in this case, the data in the first column may be moved in a single transaction to a block's shared memory. Thereupon, each thread within that block may read its own individual element from shared memory. This would then repeat for subsequent columns, one after the other.

This is an example of data coalescing, and in practice can result in very substantial impacts to the runtime, just by optimising the memory access patterns.

In a similar vein is the concern of *race conditions* (Herlihy, Shavit, Luchangco, & Spear, 2020). This is a problem which arises when multiple threads access and modify shared data simultaneously without proper synchronisation, leading to unpredictable and incorrect behaviour. A simple example follows.

Example 3. Consider the operation:

$$\text{counter} += 1 \quad (2.83)$$

given a global `counter` variable, initialised to 0. This single operation in fact

consists of three steps:

1. *Read counter.*
2. *Add 1.*
3. *Write result to counter.*

Two threads executing this kernel at the same time might proceed as follows:

1. *Thread A reads $counter = 0$.*
2. *Thread B reads $counter = 0$.*
3. *Thread A increments $counter = 0 + 1 = 1$.*
4. *Thread B increments $counter = 0 + 1 = 1$.*
5. *Thread A writes $counter = 1$.*
6. *Thread B writes $counter = 1$.*

The result in this case is $counter = 1$, rather than the expected $counter = 2$.

Such race conditions can be avoided with the use of *atomic operations* (NVIDIA, 2020). These are special versions of primitive operators which process its constituent steps as a single uninterrupted unit. For instance, `AtomicAdd` will process its **read**, **add**, and **write** steps uninterrupted, meaning whenever a thread is executing this on a particular global variable, other threads which aim to do likewise will first wait until the current thread has finished its whole operation.

Atomic operations avoid race conditions and ensure correct and predictable outcomes when parallel threads read and write the same global data. But, they are slower than their non-atomic counterparts as parallel threads essentially (to a certain degree) execute these operations in sequence rather than parallel. Fortunately, in reality, it is not quite as slow and inefficient as this implies, since tiered

caching is used to maintain much of the parallelism.

2.5.5 Data Pipelining

To gain the full benefit of the hardware, a large amount of data ought to be processed. Usually, this will mean the number of threads, t , will greatly exceed the number of parallel cores, c , available. This (to greatly oversimplify) means the threads will be processed in $\frac{t}{c}$ sequential batches of c threads. If sufficient space is available on the GPU then it is often possible, and indeed appropriate, to send (or at least begin to send) the next batch of data to the GPU while the current batch is being processed. This ensures that when the current batch is finished it will not have to wait (as long) for the next batch of data to arrive. This is known as *asynchronous memory transfer* or simply *pipelining* the data.

This is particularly effective when the kernel runtime is greater than the data transfer time for a whole batch. In this scenario, by the time the current batch of data has finished processing, the next batch is already waiting on the GPU and thus may immediately begin its processing without delay. With the exception of that of the initial batch, the data transfer time is then negligible. Indeed, as the number of threads to process is increased, the *overall* data transfer asymptotically approaches zero.

Without pipelining, only one batch of data (plus, perhaps, some intermediate results) need be stored on the GPU at any given time. *With* pipelining, up to two batches (or more in advanced pipelines) may be stored — the current one being processed and the next one waiting — at any given time. Either way, this will usually represent only a very small fraction of the total amount of data to be processed, with most of it either already completed or not yet sent from the CPU. This is helpful to note as it means huge amounts of data (10s or 100s of gigabytes

or more) that would exceed the global memory space of the GPU may still be processed upon it.

3 | Parameterised ZX-Calculus

The motivation of this chapter arises from identifying a number of areas, particularly within classical simulation, where it is necessary to simplify a potentially large set of ZX-diagrams that share a common structure and vary only in some spider phases. To this end, it will be shown how such a set of diagrams may be expressed as a single parameterised ZX-diagram and how reasoning upon this parameterised diagram can amount to reasoning upon each constituent element in the greater set.

More tangibly, this chapter introduces a generalisation of the rewriting rules of ZX-calculus to allow for a type of *parametric rewriting* that enables simplification of ZX-diagrams with Boolean free parameters. In this way, the bulk of the computational work in reducing many like-structured ZX-diagrams can be shared across all instances, with only comparatively quick computation required after the fact in order to attain all the independent results.

In fact, this final post-processing step can also be drastically sped up as the symmetry in both the data and the calculations lend themselves well to efficient GPU-based parallel processing. Ultimately, for very common classical simulation tasks, these techniques together can offer a runtime speedup up to and beyond a factor of 100 given modest commercial hardware.

This chapter is based upon, and builds upon, the work presented in (Sutcliffe & Kissinger, 2024a). Some terminology and notation has been altered from (the original version of) this paper for the sake of clarity.

3.1 Parametric Symmetry

When dealing with ZX-calculus, it is common to process many ZX-diagrams that are — at least prior to simplification — near identical aside from some localised differences. Notably, this occurs when varying qubit outputs, as in the summation method of weak simulation outlined in lemma 2, and when applying stabiliser decompositions. Indeed, the former case involves processing an exponential number of ZX-diagrams that are in fact *structurally* identical. That is to say, the graph structure, given by the set of spiders and the edge connections among them, is the same for all instances involved. The only feature that distinguishes them from one another is their respective unique basis effects, $\langle a_1, \dots, a_n |$, on their n qubit outputs. In fact, it is precisely because of this commonality that their sum is expressible as a sum over a single parameterised ZX-diagram, varying in some free Boolean parameters, $a_1, \dots, a_n \in \mathbb{B}$.

Sets of ZX-diagrams that share this kind of commonality occur frequently in classical simulation tasks especially and are the main focus of this chapter. As such, it is helpful to introduce a formal definition that describes such sets:

Definition 26. *A set of ZX-diagrams $G = \{g_1, g_2, \dots, g_n\}$ is said to be ‘parametrically symmetric’ iff:*

- *every element in G is structurally isomorphic:*

$$g_i \cong_{\text{struct}} g_j \quad \forall i, j \in \{1, 2, \dots, n\} \quad (3.1)$$

and

- *for every vertex v among the full set of vertices V in this common graph*

structure, its phase in each $g_i \in G$ is either the same or varies by $\pm\pi$:

$$\phi_{g_i}(v) = \phi_{g_j}(v) \quad \text{or} \quad \phi_{g_i}(v) = \phi_{g_j}(v) \pm \pi \quad \forall v \in V, \forall i, j \in \{1, \dots, n\} \quad (3.2)$$

where $\phi_{g_i}(v)$ is the phase of v in graph g_i .

To offer a graphical alternative definition: a set of ZX-diagrams $G = \{g_1, g_2, \dots, g_n\}$ is parametrically symmetric iff, up to spider fusion:

$$\boxed{g_i} = f(a_{i1}, a_{i2}, \dots, a_{im}) \boxed{h} \begin{matrix} \text{---} a_{i1}\pi \\ \text{---} a_{i2}\pi \\ \vdots \\ \text{---} a_{im}\pi \end{matrix} \quad \forall g_i \in G \quad (3.3)$$

given a common ZX-diagram h with a scalar coefficient given by a common function f . Here, $a_{ij} \in \{0, 1\} \quad \forall j \in \{1, 2, \dots, m\}$ are Booleans associated with the graph g_i .

Essentially, parametrically symmetric ZX-diagram sets are those expressible as a single ZX-diagram with Boolean free phases which are coefficients of π . When expressed as a single such ZX-diagram, the set may be said to be ‘parameterised’.

Notice in the above definition that no care is given to any scalar factors that the ZX-diagrams in the set may have. This is because any set of scalars can be parameterised fairly trivially. Even a set of n disparate scalars $\{A, B, C, D\}$ can be parameterised with $\log_2(n)$ Boolean parameters $\mathfrak{a}, \mathfrak{b} \in \mathbb{B}^*$ like so:

$$A^{(1-\mathfrak{a})(1-\mathfrak{b})} \cdot B^{(1-\mathfrak{a})\mathfrak{b}} \cdot C^{\mathfrak{a}(1-\mathfrak{b})} \cdot D^{\mathfrak{a}\mathfrak{b}} \quad (3.4)$$

However, this simple means of scalar parameterisation gives an expression containing n factors when parameterising n unique scalars, which largely defeats the point of parameterisation. Fortunately, in practice (and as will be seen in the sections ahead), ZX-diagrams forming parametrically symmetric sets tend not to be unrelated, and so their scalars can generally be parameterised much more concisely. Typically, most non-trivial variation among the scalar coefficients of such diagrams arise from the rewriting rules, which map any scalars according to very structured patterns. This is also helped by the fact that the scalar coefficient of any Clifford+T ZX-diagram is restricted to the set $\mathbb{D}[e^{i\pi/4}]$, where \mathbb{D} is the ring of dyadic rational numbers (Giles & Selinger, 2013): $\mathbb{D} := \mathbb{Z}[\frac{1}{2}]$.

3.2 Parameterising ZX-Calculus

Most applications of ZX-calculus deal with wholly numerical (i.e. non-parametric) ‘static’ ZX-diagrams. These can be manipulated and simplified via the rewriting rules very simply and straightforwardly. However, there are a number of applications, such as quantum machine learning (Toumi, Yeung, & de Felice, 2021; Koch, 2022; Yeung, 2020) and circuit differentiation (Q. Wang, Yeung, & Koch, 2024; Q. Wang & Yeung, 2022), in which it is necessary to deal with *parameterised* ZX-diagrams. Generally, in such situations, the parameterised parts of the ZX-diagrams are essentially abstracted and treated separately from the non-parameterised parts.

Nevertheless, as outlined above, there is also much utility, particularly in the field of classical simulation, in simplifying parameterised ZX-diagrams. Consequently, this section presents generalisations to the rewriting rules to support (appropriately restricted) parameterised phases, allowing such parameterised ZX-diagrams to be simplified and reduced as one, while maintaining generality.

By definition 26, such parameterised ZX-diagrams support spiders of the form:

where $\text{Image}(\Psi) = \{0, \pi\}$.

In allowing spider fusion, it follows that parameterised spiders of a more general form must also be permitted:

with $\text{Image}(\Psi + \alpha) = \{\alpha, \alpha + \pi\}$, where $\alpha \in \mathbb{R}$.

Such parameterised phases, of this general form, may be referred to as ‘*polarised*’:

Definition 27. A parameterised spider phase Φ is said to be polarised iff:

where $\alpha \in \mathbb{R}$.

This is so-called because the image of a polarised phase may be visualised as two polar opposite points on the *phase wheel* of figure 3.1.

These phases may be given by a set of Boolean free parameters and a constant component:

Lemma 7. *A parameterised spider phase Φ may be expressed in the form:*

$$\phi = \Phi(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n) = (\mathbf{a}_1 \oplus \mathbf{a}_2 \oplus \dots \oplus \mathbf{a}_n)\pi + \alpha \quad (3.8)$$

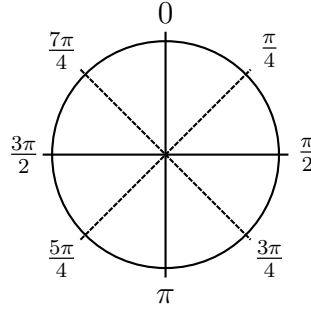


Figure 3.1: The *phase wheel*, depicting the possible numerical phases $\bmod 2\pi$ which a spider may take. Any *parameterised* spider phase must be such that its image consists of two opposite points on this wheel.

where $\alpha \in \mathbb{R}$ and $\mathbf{a}_i \in \mathbb{B}^* \forall i = 1, 2, \dots, n$. Note here that \oplus is the XOR operator (addition modulo 2) and $n \geq 1$.

Note that, due to the modulo 2π nature of phases, fusing two polarised spiders, $\text{Image}(\Psi) = \{\alpha, \alpha + \pi\}$ and $\text{Image}(\Phi) = \{\beta, \beta + \pi\}$, gives another polarised spider:

where $\text{Image}(\Psi + \Phi) = \{\alpha + \beta, \alpha + \beta + \pi\}$, given $\alpha, \beta \in \mathbb{R}$.

Naturally, such phases are not limited to Z-spiders:

3.2.2 Parameterising State Copy

While the spider fusion and colour change rules generalise for polarised phases rather trivially, most of the other rewriting rules require more consideration. In-

deed, the rewriting rules of ZX-calculus, as outlined in figures 2.2 and 2.3, depend upon specific structures and, in some cases, with specific restrictions regarding its spider phases. Consider, for example, the state copy rule¹:

$$\text{red circle } a\pi \text{ --- green circle } \alpha \text{ with two outgoing lines} = \frac{1}{\sqrt{2}} e^{ia\alpha} \begin{array}{c} \text{red circle } a\pi \text{ ---} \\ \text{red circle } a\pi \text{ ---} \end{array} \quad (3.11)$$

where $a \in \mathbb{B}$ and $\alpha \in \mathbb{R}$.

The requirements that must be met for this rule to be applicable fall into two categories: the structural requirement and the phase requirement. The former requires a 1-degree (*‘lollipop’*) spider of phase ϕ be connected via a normal edge to an opposite coloured 3-degree spider of phase ψ (or, equivalently, to a like-coloured spider via a Hadamard edge). Meanwhile, the phase requirement insists $\phi \in \{0, \pi\}$ and $\psi \in \mathbb{R}$. Together, the rule may be expressed as:

$$\text{red circle } \phi \text{ --- green circle } \psi \text{ with two outgoing lines} = \frac{1}{\sqrt{2}} e^{i\phi\psi/\pi} \begin{array}{c} \text{red circle } \phi \text{ ---} \\ \text{red circle } \phi \text{ ---} \end{array} \iff \phi \in \{0, \pi\} \quad (3.12)$$

where $\phi, \psi \in \mathbb{R}$.

Firstly, it can be assumed that all spiders take a real phase and hence the $\psi \in \mathbb{R}$ condition is necessarily met. Now suppose the lollipop spider takes a parameterised phase of $\phi = \Phi(a) = a\pi$, as might be derived from parameterising a parametrically symmetric pair of ZX-diagrams. Note here, and throughout this chapter, that $a, b, \dots \in \{0, 1\}^*$ and $a, b, \dots \in \{0, 1\}$ (see Preliminaries). While the exact phase ϕ of this spider remains unknown, its set of possible states, or its *‘image’*, is very precisely and restrictively defined, namely $\Phi(a) \in \{0, \pi\}$ (or, equivalently, $\text{Image}(\Phi) = \{0, \pi\}$). Consequently, despite the phase of the spider remain-

¹This is the $n = 2$ version of the rule, which may be used to derive the general n case.

ing unspecified, one can nevertheless say with certainty that the state copy rule does in fact apply as both the structural and phase requirements are indeed met. The same would be true given, for example, $\phi = \Phi(\mathfrak{a}, \mathfrak{b}) = \mathfrak{a}\pi + \mathfrak{b}\pi + \pi \bmod 2\pi^2$, as this too has an image of $\Phi(a, b) \in \{0, \pi\}$.

Thus, the state copy rule can be generalised (i.e. parameterised) as:

$$\text{Diagram: a green circle with a red circle inside it, labeled } \Phi \text{ and } \Psi \text{ respectively, with two wires entering from the left and two exiting to the right.} = \frac{1}{\sqrt{2}} e^{i\Phi\Psi/\pi} \text{Diagram: two red circles, each labeled } \Phi, \text{ with two wires entering from the left and two exiting to the right.} \iff \text{Image}(\Phi) \subseteq \{0, \pi\} \quad (3.13)$$

where Φ and Ψ are algebraic expressions, each containing a subset of Boolean parameters. That is, given some global set of parameters $P = \{\mathfrak{a}, \mathfrak{b}, \dots\}$ where $\mathfrak{a}, \mathfrak{b}, \dots \in \{0, 1\}$, $\Phi(X)$ and $\Psi(Y)$ are such that $X, Y \subseteq P$. Note that, as this is a generalisation of the rewriting rule, the empty set is also a valid subset, $\emptyset \subseteq P$, meaning Φ and/or Ψ may also take constant values.

This parameterised version of the state copy rule is subtly, but importantly, distinct from the conventional version. The latter (equation 3.12) should be interpreted to mean:

The rule, as expressed graphically, is valid if and only if the spider phase ϕ takes the specific value of 0 or π .

The parameterised version (equation 3.13), meanwhile, should be interpreted as:

The rule, as expressed graphically, is valid if and only if the spider phase Φ takes an algebraic expression whose image (set of possible states) is $\{0, \pi\}$, $\{0\}$, or $\{\pi\}$.

Essentially, the difference is: in the conventional version of the rule, ϕ denotes a specific real number, whereas in the parameterised version of the rule, Φ denotes

²or, equivalently, $\phi = \Phi(\mathfrak{a}, \mathfrak{b}) = (\mathfrak{a} \oplus \mathfrak{b} \oplus 1)\pi$.

an algebraic expression.

Lastly, note that a constant value, such as π , is also a valid algebraic expression, with $\text{Image}(\pi) = \{\pi\}$. As such, the parameterised version of the rule (equation 3.13) is strictly a generalisation of the conventional form (equation 3.12).

3.2.3 Parameterising the Remaining Rewriting Rules

Most of the other rewriting rules can be trivially parameterised likewise, with two exceptions. The first is identity removal:

$$\text{---} \textcircled{\phi} \text{---} = \text{---} \iff \phi = 0 \quad (3.14)$$

In attempting to parameterise this rule, one must first introduce a parameterised phase Φ with a polarised image, $\text{Image}(\Phi) = \{\alpha, \alpha + \pi\}$, to the left-hand side:

$$\text{---} \textcircled{\Phi} \text{---}$$

Rather problematically, this cannot always be resolved with certainty, despite the polarised nature of the Φ phase. Unlike (most of) the other rewriting rules, identity removal requires a specific phase, namely 0. Consequently, given a parameterised phase such as $\Phi = \alpha\pi$, with $\text{Image}(\Phi) = \{0, \pi\}$, it remains ambiguous whether the phase condition is met and hence ambiguous whether identity removal is applicable.

The second problematic rewriting rule is π -commutation:

$$\begin{array}{c} \text{---} \textcircled{\phi} \text{---} \psi \text{---} \\ \vdots \end{array} = e^{i\psi} \begin{array}{c} \text{---} \textcircled{-\psi} \text{---} \\ \textcircled{\phi} \\ \textcircled{\phi} \\ \vdots \\ \textcircled{\phi} \end{array} \iff \psi = \pi \quad (3.15)$$

though — as it turns out — only under certain conditions. The more general expression of the rule:

$$\begin{array}{c} \text{---} \phi \text{---} \psi \text{---} \vdots \end{array} = e^{i\phi\psi/\pi} \begin{array}{c} \phi \\ \phi \\ (-1)^{\phi/\pi\psi} \vdots \\ \phi \end{array} \iff \phi \in \{0, \pi\} \quad (3.16)$$

gives a stronger starting point. However, attempting to parameterise this naïvely:

$$\begin{array}{c} \text{---} \Phi \text{---} \Psi \text{---} \vdots \end{array} = e^{i\Phi\Psi/\pi} \begin{array}{c} \Phi \\ \Phi \\ (-1)^{\Phi/\pi\Psi} \vdots \\ \Phi \end{array} \iff \text{Image}(\Phi) \subseteq \{0, \pi\} \quad (3.17)$$

leads to $\text{Image}(\Phi) = \{0, \pi\}$ and $\text{Image}((-1)^{\Phi/\pi\Psi}) = \{\alpha, -\alpha\}$. The problem lies in the fact that the latter is, in general, not polarised (unless $\alpha \in \{\frac{\pi}{2}, -\frac{\pi}{2}\}$) and, by definition 26, all parameterised phases must be polarised to maintain parametric symmetry.

By spider fusion, allowing such phases means allowing wholly arbitrary parameterised spiders:

$$\begin{array}{c} \vdots \\ \Psi \end{array} \begin{array}{c} \vdots \\ \Phi \end{array} = \begin{array}{c} \vdots \\ \Psi + \Phi \end{array} \quad (3.18)$$

where $\text{Image}(\Phi) = \{\alpha, -\alpha\}$ and $\text{Image}(\Psi) = \{\beta, -\beta\}$, and hence:

$$\text{Image}(\Phi + \Psi) = \{\alpha + \beta, \alpha - \beta, -\alpha + \beta, -\alpha - \beta\}.$$

Ultimately, this leads to phases Φ with $\text{Image}(\Phi) = \mathbb{R}$. Allowing such phases

would violate definition 26 by breaking the parametric symmetry.

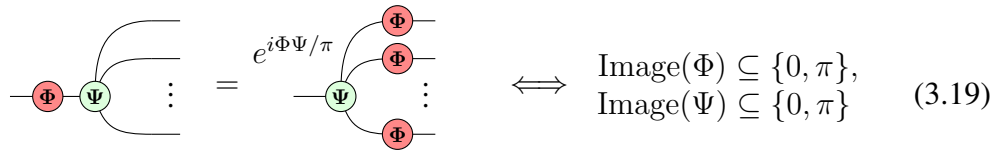
So, in order to maintain parametric symmetry through rewriting, this general parameterisation of the π -commutation rule cannot be permitted. Nevertheless, special cases of the rule can be. Recall equation 3.17 and consider restricting $\text{Image}(\Psi) = \{0, \pi\}$. In this special case, the phase mapping:

$$\Psi \rightarrow (-1)^{\Phi/\pi} \Psi$$

is equivalent to:

$$\Psi \rightarrow \Psi$$

since $-0 = 0$ and $-\pi = \pi$. As a result, this special case of the rule can be parameterised:



$$\text{Diagram} = e^{i\Phi\Psi/\pi} \text{Diagram} \iff \begin{aligned} \text{Image}(\Phi) &\subseteq \{0, \pi\}, \\ \text{Image}(\Psi) &\subseteq \{0, \pi\} \end{aligned} \quad (3.19)$$

while maintaining polarisation on all parameterised phases. As another special case, consider once more equation 3.17, this time with the restriction $\text{Image}(\Psi) = \{\frac{\pi}{2}, -\frac{\pi}{2}\}$. Under this condition, the phase mapping:

$$\Psi \rightarrow (-1)^{\Phi/\pi} \Psi$$

is equivalent to:

$$\Psi \rightarrow \Psi + \Phi$$

as $-\frac{\pi}{2} + \pi = \frac{\pi}{2}$ and $\frac{\pi}{2} + \pi = -\frac{\pi}{2}$. This gives a second valid special case parame-

terisation of the π -commutation rule:

$$\begin{array}{c} \text{---} \Phi \Psi \text{---} \\ \vdots \end{array} = e^{i\Phi\Psi/\pi} \begin{array}{c} \text{---} \Psi + \Phi \text{---} \\ \vdots \end{array} \iff \begin{array}{l} \text{Image}(\Phi) \subseteq \{0, \pi\}, \\ \text{Image}(\Psi) \subseteq \{\frac{\pi}{2}, -\frac{\pi}{2}\} \end{array} \quad (3.20)$$

Between these two cases, the π -commutation rule has been fully parameterised for the Clifford set, whereby every phase $\phi_i \forall i$ is either:

$$\phi_i = \frac{n\pi}{2}$$

or:

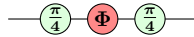
$$\phi_i = \Phi_i(\mathbf{a}_{i1}, \mathbf{a}_{i2}, \dots, \mathbf{a}_{im}), \quad \text{where } \text{Image}(\Phi_i) = \left\{ \frac{n\pi}{2}, \frac{n\pi}{2} + \pi \right\},$$

given $n \in \{0, 1, 2, 3\}$ and $\mathbf{a}_{ij} \in^* \{0, 1\} \forall j \in \{1, 2, \dots, m\}$.

Unfortunately, beyond the Clifford regime, it is not possible to parameterise the π -commutation rule (equation 3.17) without breaking the parametric symmetry:

$$\text{Image}((-1)^{\Phi/\pi}\Psi) = \{\alpha, \alpha + \pi\} \iff \text{Image}(\Psi) \in \left\{ \{0, \pi\}, \left\{ \frac{\pi}{2}, -\frac{\pi}{2} \right\} \right\} \quad (3.21)$$

where $\alpha \in \mathbb{R}$. This means when incorporating the wider Clifford+T gateset, ZX-diagrams such as:



where, for instance, $\phi = \Phi(\mathbf{a}, \mathbf{b}) = (\mathbf{a} \oplus \mathbf{b})\pi$, cannot be simplified, despite the fact that for either possible state of $\phi \in \{0, \pi\}$ this diagram would be reducible to Clifford. In other words, the two possible reduced states:

and

cannot be expressed as a single parameterised ZX-diagram with Boolean free parameters. That is to say, the set $\left\{ \textcircled{\pi/2}, e^{i\pi/4} \textcircled{\pi} \right\}$ is not parametrically symmetric.

Furthermore, at first glance, it would appear that the bialgebra rule:

runs into the same problem faced by identity removal, as this too insists upon specific phases (again, of 0). So, given parameterised phases, Φ and Ψ , such that $\text{Image}(\Phi) = \{0, \pi\}$ and $\text{Image}(\Psi) = \{0, \pi\}$, it is apparently ambiguous whether bialgebra may apply. In fact, this issue can be resolved by first unfusing the parameterised phases and performing non-parameterised bialgebra:

From here, the parameterised π -commutation rule, introduced in equation 3.19, allows Φ and Ψ to switch sides:

Thus, a parameterised bialgebra rule has been derived:

$$\begin{array}{c} \text{---} \Phi \text{---} \Psi \text{---} \end{array} = \sqrt{2} e^{i\Phi\Psi/\pi} \begin{array}{c} \text{---} \Psi \text{---} \Phi \text{---} \\ \text{---} \Psi \text{---} \Phi \text{---} \end{array} \iff \begin{array}{l} \text{Image}(\Phi) \subseteq \{0, \pi\}, \\ \text{Image}(\Psi) \subseteq \{0, \pi\} \end{array} \quad (3.25)$$

The culmination of the above is a full set of parameterised rewriting rules, as collected in figure 3.2, where every phase is either a specific real value or a polarised algebraic expression:

$\forall \vartheta \in \{\psi, \phi\} :$

- $\vartheta \in \mathbb{R}$, or
- $\vartheta = \Theta(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)$ with $\text{Image}(\Theta) = \{\alpha, \alpha + \pi\}$, given $\alpha \in \mathbb{R}$ and $\mathbf{a}_i \stackrel{*}{\in} \mathbb{B} \ \forall i \in \{1, 2, \dots, n\}$.

Some higher level rewriting rules derived from the basic set can also be parameterised, such as local complementation and pivoting (see figure 2.3), as shown in figure 3.3.

The rules expressed in figures 3.2 and 3.3 may be used to simplify any parameterised (or indeed static) ZX-diagram. With the exception of identity removal, these rules are complete for parameterised Clifford ZX-diagrams, meaning any such diagram is reducible to the same extent as its static counterparts (except for any leftover 2-legged polarised spiders, which may be pushed to one side via π -commutation). Any *scalar* parameterised Clifford ZX-diagram is thus fully reducible to a parameterised scalar.

However, for parameterised Clifford+T ZX-diagrams, the π -commutation rule is not fully complete, meaning that there are circumstances where it may not be applied, despite it being applicable in the static equivalents of the given diagram. Specifically, a static phase $\phi \in \{0, \pi\}$ may commute through a T-phase (static

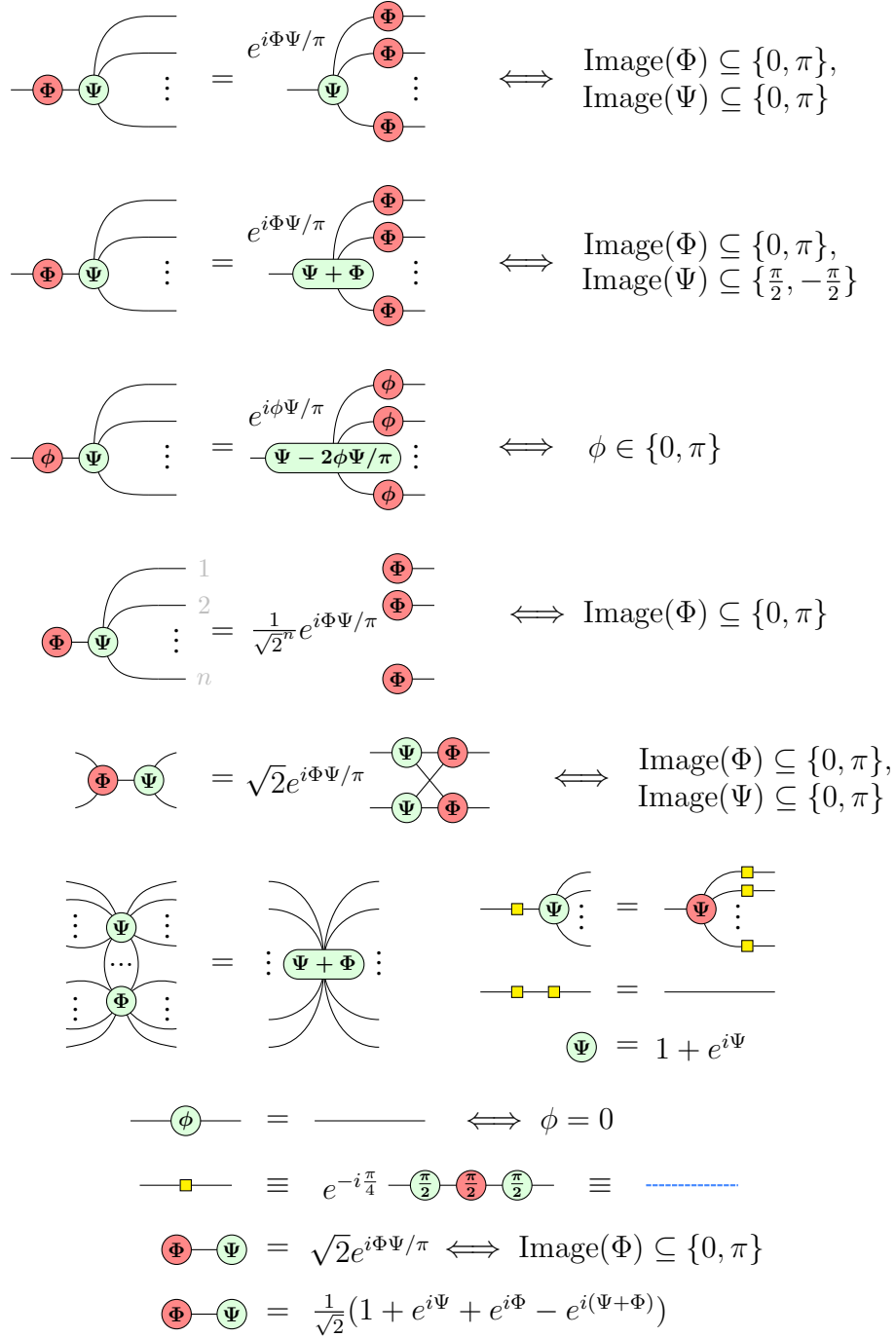


Figure 3.2: The complete set of parameterised rewriting rules, where Ψ and Φ are parameterised or static phases such that $\text{Image}(\Psi) \subseteq \{\alpha, \alpha + \pi\}$ and $\text{Image}(\Phi) \subseteq \{\beta, \beta + \pi\}$, given $\alpha, \beta \in \mathbb{R}$. Meanwhile, $\phi \in \mathbb{R}$ is strictly static.

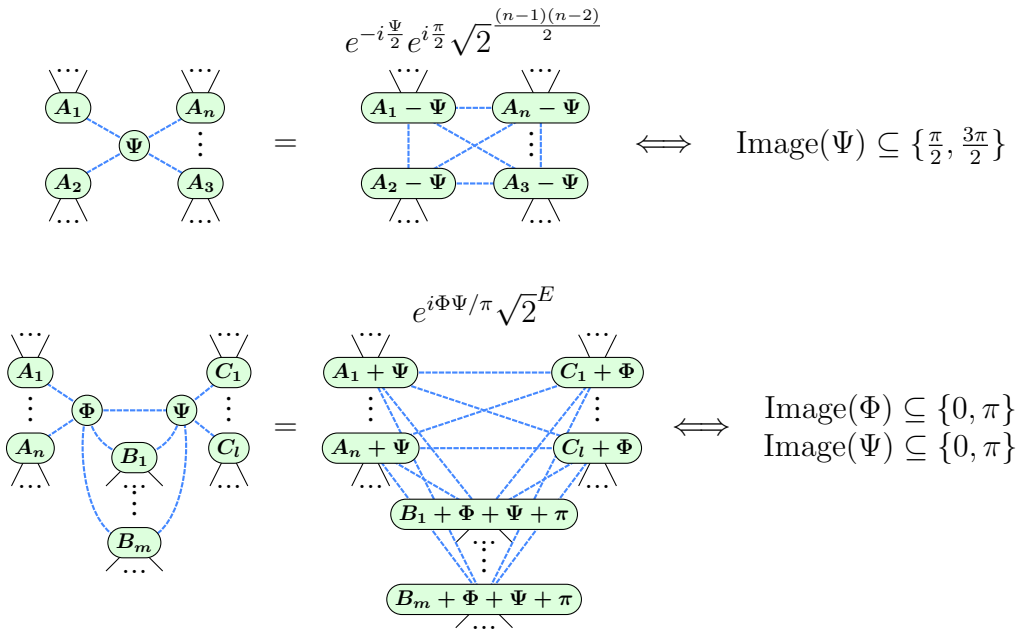


Figure 3.3: Parameterised local complementation and pivoting rules, where $E = (n-1)m + (l-1)m + (n-1)(l-1)$ and each $\Theta \in \{\Psi, \Phi, A_1, \dots, A_n, B_1, \dots, B_m, C_1, \dots, C_l\}$ is a parameterised or static phase such that $\text{Image}(\Theta) \subseteq \{\alpha_\Theta, \alpha_\Theta + \pi\}$, given $\alpha_\Theta \in \mathbb{R} \forall \Theta$.

or otherwise), such as $\frac{\pi}{4}$, without issue, whereas a parameterised phase Φ with $\text{Image}(\Phi) = \{0, \pi\}$ is *not* able to commute through such a T-spider without breaking parametric symmetry, due to equation 3.21. Consequently, a parameterised Clifford+T ZX-diagram might not reduce as far as its static equivalents. The consequence of this is emphasised in lemma 8. Despite this, as will be seen, this is seldom a significant issue in practice, with parameterised Clifford+T ZX-diagrams being reducible to *almost* the same extent as their static counterparts.

Lemma 8. *Simplifying a parameterised scalar Clifford+T ZX-diagram will always result in at least as many stabiliser terms as would be attained from simplifying the equivalent static (i.e. non-parameterised) diagram.*

3.2.4 Parameterised Scalar Expressions

After fully decomposing and simplifying a parameterised scalar ZX-diagram, one is left with a parameterised expression denoting its potential scalar values:

Lemma 9. *A parameterised scalar ZX-diagram is reducible to a parameterised expression $S(\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n) \in \mathbb{C}^*$:*

$$\begin{array}{c} \boxed{G} \end{array} \begin{array}{c} \text{---} \textcircled{\mathfrak{a}_1 \pi} \\ \text{---} \textcircled{\mathfrak{a}_2 \pi} \\ \vdots \\ \text{---} \textcircled{\mathfrak{a}_n \pi} \end{array} \rightarrow S(\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n) \in \mathbb{C}^* \quad (3.26)$$

where $\mathfrak{a}_i \in \mathbb{B}^* \ \forall i \in \{1, 2, \dots, n\}$.

This resulting parameterised expression, S , acting as a function, may be evaluated for any bitstring input, $S(a_1, a_2, \dots, a_n)$ where $a_i \in \mathbb{B} \ \forall i \in \{1, 2, \dots, n\}$, to obtain the scalar result of the corresponding ZX-diagram instance, such that the

following equality holds:

$$G \begin{matrix} \text{---} a_1 \pi \\ \text{---} a_2 \pi \\ \vdots \\ \text{---} a_n \pi \end{matrix} = S(a_1, a_2, \dots, a_n) \in \mathbb{C} \quad (3.27)$$

Given lemma 9 and a parameterised scalar ZX-diagram containing n Boolean free parameters, $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{B}$, one may deduce all 2^n scalar values (one for each input bitstring) via just one instance of ZX-calculus reduction plus 2^n evaluations of the resulting parameterised scalar. This is as opposed to 2^n full reductions of unique ZX-diagrams, as would be required without parameterisation.

Importantly, the resulting parameterised scalar always follows a particular structure, being a sum of terms, where each term is the product of (a constant and) one or more parameterised *subterms*:

$$S = \sum_{i=1}^m \left[C_i \prod_{j=1}^{n_i} s_{ij} \right] \quad (3.28)$$

Here, s_{ij} denotes the j^{th} subterm of the i^{th} term of the parameterised scalar S . m is then the number of terms and n_i is the number of subterms in term i . Lastly, C_i is a constant (i.e. non-parameterised) factor associated with term i . Figure 3.4 shows a visual breakdown of this structure.

From the parameterised rewriting rules and scalar relations presented in the previous subsection, one may observe that there are in fact four different types of parameterised subterms that may arise. These are summarised in table 3.1.

In fact, it can be shown that these may all reduce to a single unique subterm type,

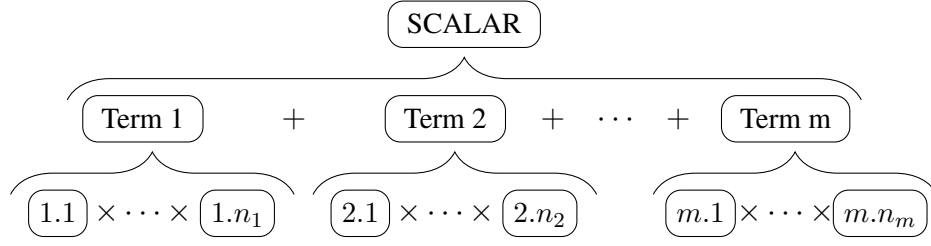


Figure 3.4: A breakdown of the structure of a parameterised scalar. A scalar Clifford+T ZX-diagram may be reduced to a parameterised scalar expression, comprised of a sum of *terms*, with each term being comprised of a product of *subterms* (and a constant).

Name	Form	Origin(s)
Node	$(1 + e^{i\Psi})$	
Phase-pair	$(1 + e^{i\Psi} + e^{i\Phi} - e^{i(\Psi+\Phi)})$	
Half- π	$e^{i\Psi/2}$	Figure 3.3
π -pair	$e^{i\Psi\Phi/\pi}$	Figures 3.2 and 3.3

Table 3.1: The different types of subterms that may arise from reducing parameterised ZX-diagrams, where Ψ and Φ are parameterised phases obeying the form of lemma 7.

namely that labelled ‘*phase-pair*’. Lemmas 10 to 12 formalise this observation.

Lemma 10. *Node-type subterms can be reduced to phase-pair subterms:*

$$(1 + e^{i\Psi}) = C \left(1 + e^{i\Psi'} + e^{i\Phi} - e^{i(\Psi'+\Phi)} \right) \quad (3.29)$$

where $\Psi' = \Psi + \frac{\pi}{2}$ and $C = \frac{\sqrt{2}}{4}(1 - i)$.

Proof.

$$\begin{aligned}
 (1 + e^{i\Psi}) &= \textcircled{\Psi} & (3.30) \\
 &= \textcircled{\Psi + \frac{\pi}{2}} \text{---} \textcircled{-\frac{\pi}{2}} \\
 &= \textcircled{\Psi + \frac{\pi}{2}} \text{---} \textcircled{-\frac{\pi}{2}} \\
 &= e^{-i\frac{\pi}{4}} \textcircled{\Psi + \frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \textcircled{-\frac{\pi}{2}} \\
 &= e^{-i\frac{\pi}{4}} \textcircled{\Psi + \frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \textcircled{\phantom{\frac{\pi}{2}}} \\
 &= \frac{1}{\sqrt{2}} e^{-i\frac{\pi}{4}} \textcircled{\Psi + \frac{\pi}{2}} \textcircled{\frac{\pi}{2}} \\
 &= \frac{1}{\sqrt{2}} e^{-i\frac{\pi}{4}} \cdot \frac{1}{\sqrt{2}} (1 + e^{i(\Psi + \frac{\pi}{2})} + e^{i\Phi} - e^{i(\Psi + \frac{\pi}{2} + \Phi)}) \\
 &= \frac{\sqrt{2}}{4} (1 - i) (1 + e^{i\Psi'} + e^{i\Phi} - e^{i(\Psi' + \Phi)})
 \end{aligned}$$

□

Lemma 11. π -pair subterms can be reduced to phase-pair subterms:

$$e^{i\Psi\Phi/\pi} \rightarrow C (1 + e^{i\Psi} + e^{i\Phi} - e^{i(\Psi+\Phi)}) \quad (3.31)$$

where $C = \frac{1}{2}$.

Proof. There are five sources from which π -pair subterms, $e^{i\Psi\Phi/\pi}$, may arise, being:

- parameterised state copy (figure 3.13),
- parameterised pivoting (figure 3.3),
- parameterised π -commutation (figures 3.19 and 3.20),
- parameterised bialgebra (figure 3.25), and

- parameterised special case phase-pair (figure 3.2).

In each case, one may observe that $\text{Image}(\Phi) \subseteq \{0, \pi\}$. Under this restriction, the relation holds:

$$e^{i\Psi\Phi/\pi} = \frac{1}{2} (1 + e^{i\Psi} + e^{i\Phi} - e^{i(\Psi+\Phi)}) \quad (3.32)$$

Explicitly, if $\Phi = 0$:

$$e^0 = \frac{1}{2} (1 + e^{i\Psi} + e^0 - e^{i(\Psi+0)}) \quad (3.33)$$

$$1 = \frac{1}{2} (1 + 1)$$

$$1 = 1$$

$$\therefore \text{LHS} = \text{RHS}$$

Likewise, if $\Phi = \pi$:

$$e^{i\Psi} = \frac{1}{2} (1 + e^{i\Psi} + e^{i\pi} - e^{i(\Psi+\pi)}) \quad (3.34)$$

$$e^{i\Psi} = \frac{1}{2} (1 + e^{i\Psi} + (-1) - e^{i\Psi} e^{i\pi})$$

$$e^{i\Psi} = \frac{1}{2} (1 + e^{i\Psi} - 1 + e^{i\Psi})$$

$$e^{i\Psi} = \frac{1}{2} (2e^{i\Psi})$$

$$e^{i\Psi} = e^{i\Psi}$$

$$\therefore \text{LHS} = \text{RHS}$$

Hence, graphically:

$$\textcircled{\Psi} \text{---} \textcircled{\Phi} = \sqrt{2} e^{i\Psi\Phi/\pi} \quad (3.35)$$

provided $\text{Image}(\Phi) \subseteq \{0, \pi\}$. □

Lemma 12. *Half- π subterms can be reduced to phase-pair subterms:*

$$e^{i\Psi/2} \rightarrow C \left(1 + e^{i\Psi'} + e^{i\Phi} - e^{i(\Psi'+\Phi)} \right) \quad (3.36)$$

where $\Psi' = -\Psi + \frac{\pi}{2}$ and $C = \frac{1}{2}$.

Proof. Half- π subterms arise from instances of parameterised local complementation (figure 3.3), from which it may be observed that $\text{Image}(\Psi) \subseteq \{\frac{\pi}{2}, \frac{3\pi}{2}\}$.

These terms may be slightly rewritten with a change of variable:

$$e^{-i\frac{\Psi}{2}} e^{i\frac{\pi}{2}} = e^{\frac{i}{2}(-\Psi+\frac{\pi}{2})} e^{\frac{i\pi}{4}} = e^{\frac{i}{2}\Psi'} e^{\frac{i\pi}{4}} \quad (3.37)$$

where $\Psi' = -\Psi + \frac{\pi}{2}$ such that $\text{Image}(\Psi') \subseteq \{0, \pi\}$.

Furthermore:

$$e^{i\Psi'/2} \equiv e^{i\Psi'\Phi/\pi} \quad (3.38)$$

where $\Phi = \frac{\pi}{2}$.

Hence, the half- π subterm type is a special case of the π -pair type, which was shown in lemma 11 to be reducible to the phase-pair type. \square

Given this, the parameterised scalar expressions are always expressible in a very consistent format:

Lemma 13. *Any parameterised scalar expression arising from reducing a parameterised scalar ZX-diagram, $G(\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n)$, may be expressed according to equation 3.28, where:*

$$s_{ij} = \left(1 + e^{i\Psi_{ij}} + e^{i\Phi_{ij}} - e^{i(\Psi_{ij}+\Phi_{ij})} \right) \propto \begin{array}{c} \textcircled{\Psi_{ij}} \\ | \\ \textcircled{\Phi_{ij}} \end{array} \quad \forall i, j \quad (3.39)$$

given:

$$\begin{aligned}\Psi_{ij} &= \alpha_{ij} + \pi \bigoplus_{\mathfrak{p} \in P_{ij}^\psi} \mathfrak{p} \\ \Phi_{ij} &= \beta_{ij} + \pi \bigoplus_{\mathfrak{p} \in P_{ij}^\phi} \mathfrak{p}\end{aligned}\tag{3.40}$$

where:

$$\alpha_{ij}, \beta_{ij} \in \mathbb{R} \quad \forall i, j \tag{3.41}$$

and:

$$P_{ij}^\psi, P_{ij}^\phi \subseteq \{\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n\} \quad \forall i, j \tag{3.42}$$

Only minimal runtime reduction is achieved at this stage as evaluating a long complex expression containing many terms and subterms is, in most cases, approximately as slow as reducing the corresponding ZX-diagram. In other words, computing 2^n *evaluations* of the resulting parameterised scalar is generally comparable in speed with 2^n *simplifications* of the original ZX-diagram.

However, with the parameterised ZX-diagram reduced to a parameterised scalar expression, whose terms all conform to a consistent format, evaluations of this scalar can be very efficiently parallelised and even take advantage of the GPU.

3.3 GPU-Parallelised Evaluation

Each evaluation of such a parameterised scalar involves computing many identically structured subterms. This meets the SIMD requirement of GPU parallelism, discussed in section 2.5. By translating the data into a more primitive data structure and avoiding calculations involving complex exponentials, the requirement for simple instructions may also be met, as will be seen ahead.

3.3.1 Condensing the Data Structure

Specifically, as per lemma 13, any subterm, $s_{ij}(\Psi_{ij}, \Phi_{ij})$, may be concisely recorded as:

- The set of parameters in Ψ_{ij} : $p_{ij}^\psi \subseteq P$
- The constant part of Ψ_{ij} : $\alpha_{ij} \in \mathbb{R}$
- The set of parameters in Φ_{ij} : $p_{ij}^\phi \subseteq P$
- The constant part of Φ_{ij} : $\beta_{ij} \in \mathbb{R}$

where $P = \{a, b, c, \dots\}$ is the full set of parameters in the scalar.

Hence, any parameterised scalar expression may be expressed concisely by recording this data for each subterm (i.e. $\forall i, j$), together with the following metadata:

- The number of terms, m
- The constant factor of each term, $C_i \forall i \in \{1, 2, \dots, m\}$
- The number of subterms in each term, $n_i \forall i \in \{1, 2, \dots, m\}$

Furthermore, for reasons that will become apparent, it is beneficial for every term to contain the same number of subterms, $n_i = n \forall i$ where $n = \max_i(n_i)$. This can be enforced by padding each term with ‘dummy’ subterms which equal unity, and including a flag in each subterm to record whether it is a genuine subterm or a dummy.

Given this, a parameterised scalar expression may be recorded as a $(2nm) \times (2p + 3)$ matrix of 1-byte elements, where m and n are respectively the number of terms and the number of subterms per term, and p is the total number of parameters, $p = |P|$.

An example follows:

Example 4. *As per lemma 13, a parameterised scalar ZX-diagram, $G(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d})$, may reduce to the following parameterised scalar expression:*

$$\begin{aligned}
 g(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}) = & C_1 \left[\left(1 + e^{i\Psi_{1,1}} + e^{i\Phi_{1,1}} - e^{i(\Psi_{1,1} + \Phi_{1,1})} \right) \right. \\
 & \times \left(1 + e^{i\Psi_{1,2}} + e^{i\Phi_{1,2}} - e^{i(\Psi_{1,2} + \Phi_{1,2})} \right) \\
 & \times \left. \left(1 + e^{i\Psi_{1,3}} + e^{i\Phi_{1,3}} - e^{i(\Psi_{1,3} + \Phi_{1,3})} \right) \right] \\
 & + C_2 \left[\left(1 + e^{i\Psi_{2,1}} + e^{i\Phi_{2,1}} - e^{i(\Psi_{2,1} + \Phi_{2,1})} \right) \right] \\
 & + C_3 \left[\left(1 + e^{i\Psi_{3,1}} + e^{i\Phi_{3,1}} - e^{i(\Psi_{3,1} + \Phi_{3,1})} \right) \right. \\
 & \times \left. \left(1 + e^{i\Psi_{3,2}} + e^{i\Phi_{3,2}} - e^{i(\Psi_{3,2} + \Phi_{3,2})} \right) \right]
 \end{aligned} \tag{3.43}$$

which may alternatively be expressed as:

$$g(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}) = \frac{C_1}{\sqrt{2}^3} \begin{array}{c} \textcircled{\Psi_{1,1}} \textcircled{\Psi_{1,2}} \textcircled{\Psi_{1,3}} \\ | \quad | \quad | \\ \textcircled{\Phi_{1,1}} \textcircled{\Phi_{1,2}} \textcircled{\Phi_{1,3}} \end{array} + \frac{C_2}{\sqrt{2}^1} \begin{array}{c} \textcircled{\Psi_{2,1}} \\ | \\ \textcircled{\Phi_{2,1}} \end{array} + \frac{C_3}{\sqrt{2}^2} \begin{array}{c} \textcircled{\Psi_{3,1}} \textcircled{\Psi_{3,2}} \\ | \quad | \\ \textcircled{\Phi_{3,1}} \textcircled{\Phi_{3,2}} \end{array} \tag{3.44}$$

where:

- in term 1:

$$\diamond C_1 = \frac{1}{32}$$

$$\diamond \Psi_{1,1} = \mathfrak{a}\pi + \mathfrak{b}\pi + \frac{\pi}{2}$$

$$\diamond \Phi_{1,1} = \mathfrak{b}\pi + \mathfrak{c}\pi + \mathfrak{d}\pi + \frac{7\pi}{4}$$

$$\diamond \Psi_{1,2} = \mathfrak{c}\pi + \pi$$

$$\diamond \Phi_{1,2} = \mathfrak{a}\pi + \mathfrak{d}\pi + \frac{3\pi}{4}$$

$$\diamond \Psi_{1,3} = \mathfrak{a}\pi + \mathfrak{b}\pi + \mathfrak{c}\pi + \mathfrak{d}\pi + \frac{7\pi}{4}$$

$$\diamond \Phi_{1,3} = \mathfrak{a}\pi + \mathfrak{c}\pi + \mathfrak{d}\pi + \frac{3\pi}{2}$$

• *in term 2:*

$$\diamond C_2 = \frac{\sqrt{2}}{8} - \frac{i}{16}$$

$$\diamond \Psi_{2,1} = \mathfrak{b}\pi + \mathfrak{c}\pi + \frac{3\pi}{2}$$

$$\diamond \Phi_{2,1} = \mathfrak{a}\pi + \mathfrak{b}\pi + \mathfrak{d}\pi + \frac{3\pi}{4}$$

• *in term 3:*

$$\diamond C_3 = \frac{1}{4}$$

$$\diamond \Psi_{3,1} = \mathfrak{c}\pi + \mathfrak{d}\pi$$

$$\diamond \Phi_{3,1} = \mathfrak{a}\pi + \frac{\pi}{2}$$

$$\diamond \Psi_{3,2} = \mathfrak{b}\pi + \mathfrak{c}\pi + \frac{\pi}{2}$$

$$\diamond \Phi_{3,2} = \mathfrak{a}\pi + \mathfrak{c}\pi + \frac{\pi}{4}$$

This may then be recorded in matrix form as in table 3.2, with C_1, C_2, C_3 recorded separately among the metadata, together with $p = 4$, $m = 3$, and $n = 3$ recording, respectively, the number of parameters, terms, and subterms per term.

Here, each row represents a single subterm (i.e. a particular s_{ij}), and each set of $n = 3$ rows represents a whole term, with the column headers denoting the following:

- *The ‘*’ column records whether the subterm is genuine (1) or a dummy for padding (0).*

Table 3.2: An example of a parametric scalar expression in matrix form.

*	Ψ					Φ				
	$4\alpha/\pi$	\mathfrak{a}_ψ	\mathfrak{b}_ψ	\mathfrak{c}_ψ	\mathfrak{d}_ψ	$4\beta/\pi$	\mathfrak{a}_ϕ	\mathfrak{b}_ϕ	\mathfrak{c}_ϕ	\mathfrak{d}_ϕ
1	2	1	1	0	0	7	0	1	1	1
1	4	0	0	1	0	3	1	0	0	1
1	7	1	1	1	1	6	1	0	1	1
1	6	0	1	1	0	3	1	1	0	1
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	2	1	0	0	0
1	2	0	1	1	0	1	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0

- The ‘ $4\alpha/\pi$ ’ column records the constant part (α) of Ψ_{ij} for the subterm s_{ij} . This is multiplied by $\frac{4}{\pi}$ such that (within the Clifford+T regime) the result may be recorded as an integer in the range $[0, 7]$.
- The subsequent $p = 4$ columns, labelled \mathfrak{a}_ψ to \mathfrak{d}_ψ record whether each of these variables (\mathfrak{a} to \mathfrak{d}) in turn are (1) or are not (0) included within Ψ_{ij} for the subterm s_{ij} . In this way, Ψ may be expressed as $\Psi = (\mathfrak{a}_\psi a \oplus \mathfrak{b}_\psi b \oplus \mathfrak{c}_\psi c \oplus \mathfrak{d}_\psi d)\pi + \alpha$, where $a, b, c, d \in \mathbb{B}$ are the parameter values to evaluate.
- The final columns are analogous to the previous $p + 1 = 5$, except relating to Φ_{ij} rather than Ψ_{ij} .

Expressing parameterised scalars in this way ensures a concise and organised data structure, which will be essential to take advantage of GPU parallelism. To this end, there are some additional considerations, which are covered ahead.

3.3.2 Further Considerations

As a justification for row padding, consider the matrix of example 4, recording the subterm data of a particular parameterised scalar. For convenience, this has

been visualised with clear distinctions between each block of 3 rows, to clearly separate each term. As far as the software is concerned, however, this is simply recorded as a single $(2nm) \times (2p + 3) = 9 \times 11$ matrix of 1-byte elements, with no such demarcation between the distinct terms.

Given there may be a variable number of subterms in each term, it is helpful to include padding of dummy rows, as described, to ensure consistency. This ensures that, given the knowledge of $n = 3$, the GPU will be able to easily identify each distinct block term without additional cumbersome indexing and can compute the same sets of calculations among each, further aiding in the efficiency of the GPU-parallelism.

Furthermore, it is a small but important point that these matrices should be stored in column-major order. Consider again the data of example 4, expressed in its matrix form. For practical purposes, it is helpful to interpret this as a two-dimensional data structure, but within memory it is necessarily stored linearly. As each row is to be processed in parallel, with each successive column being processed at a single time-step, storing this in column-major order ensures the data is coalesced (as explained in more detail in section 2.5.4). This seemingly subtle consideration has a drastic impact on the runtime.

The third and final technical consideration to aid performance on the GPU is a combination of data pipelining and batching, as discussed in section 2.5.5. Beyond the most trivial cases, the total number of subterms (rows) to process will exceed the number of available parallel threads manyfold. Given this fact, there is no benefit in waiting for the entire dataset to be loaded onto the GPU before the processing commences. In fact, by pipelining the data, its transfer time to the GPU becomes negligible (except in trivially small cases). Similarly, if the size of the data exceeds the space available on the GPU (which is generally just a few

gigabytes) then processing the data in batches ensures that this does not pose a problem.

3.3.3 Computing the Subterms

To evaluate a parameterised scalar, $g(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \dots)$, expressed in matrix form on the GPU, for a particular input, $(\mathfrak{a} = a, \mathfrak{b} = b, \mathfrak{c} = c, \dots)$, where $a, b, c, \dots \in \mathbb{B}$, there are three broad steps to follow:

1. Reduce each subterm to a complex scalar for the given input.
2. Multiply together every subterm within each term.
3. Sum together the terms.

GPU parallelism may be utilised in each of these three steps, as will be seen.

With (at least the first batch of) the data on the GPU, the first step is to evaluate all the subterms for the given input, as demonstrated in example 5.

Example 5. *This step of evaluation amounts to reducing:*

$$\begin{aligned}
g(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}) = & C_1 \left[\left(1 + e^{i\pi(\mathfrak{a}+\mathfrak{b}+\frac{1}{2})} + e^{i\pi(\mathfrak{b}+\mathfrak{c}+\mathfrak{d}+\frac{7}{4})} - e^{i\pi(\mathfrak{a}+\mathfrak{b}+\frac{1}{2}+\mathfrak{b}+\mathfrak{c}+\mathfrak{d}+\frac{7}{4})} \right) \right. \\
& \times \left(1 + e^{i\pi(\mathfrak{c}+1)} + e^{i\pi(\mathfrak{a}+\mathfrak{d}+\frac{3}{4})} - e^{i\pi(\mathfrak{c}+1+\mathfrak{a}+\mathfrak{d}+\frac{3}{4})} \right) \\
& \times \left. \left(1 + e^{i\pi(\mathfrak{a}+\mathfrak{b}+\mathfrak{c}+\mathfrak{d}+\frac{7}{4})} + e^{i\pi(\mathfrak{a}+\mathfrak{c}+\mathfrak{d}+\frac{3}{2})} - e^{i\pi(\mathfrak{a}+\mathfrak{b}+\mathfrak{c}+\mathfrak{d}+\frac{7}{4}+\mathfrak{a}+\mathfrak{c}+\mathfrak{d}+\frac{3}{2})} \right) \right] \\
& + C_2 \left[\left(1 + e^{i\pi(\mathfrak{b}+\mathfrak{c}+\frac{3}{2})} + e^{i\pi(\mathfrak{a}+\mathfrak{b}+\mathfrak{d}+\frac{3}{4})} - e^{i\pi(\mathfrak{b}+\mathfrak{c}+\frac{3}{2}+\mathfrak{a}+\mathfrak{b}+\mathfrak{d}+\frac{3}{4})} \right) \right] \\
& + C_3 \left[\left(1 + e^{i\pi(\mathfrak{c}+\mathfrak{d})} + e^{i\pi(\mathfrak{a}+\frac{1}{2})} - e^{i\pi(\mathfrak{c}+\mathfrak{d}+\mathfrak{a}+\frac{1}{2})} \right) \right. \\
& \times \left. \left(1 + e^{i\pi(\mathfrak{b}+\mathfrak{c}+\frac{1}{2})} + e^{i\pi(\mathfrak{a}+\mathfrak{c}+\frac{1}{4})} - e^{i\pi(\mathfrak{b}+\mathfrak{c}+\frac{1}{2}+\mathfrak{a}+\mathfrak{c}+\frac{1}{4})} \right) \right]
\end{aligned} \tag{3.45}$$

to, for instance:

$$\begin{aligned}
g(0, 1, 1, 0) = & C_1 (1 + \sqrt{2} - i) (2) (1 + i - i\sqrt{2}) \\
& + C_2 (1 + \sqrt{2} - i) \\
& + C_3 (2i) (1 - \sqrt{2} + i)
\end{aligned} \tag{3.46}$$

where $C_1, C_2, C_3 \in \mathbb{C}$, such as:

$$\begin{aligned}
C_1 &= \frac{1}{32} \\
C_2 &= \frac{\sqrt{2}}{8} - \frac{i}{16} \\
C_3 &= \frac{1}{4}
\end{aligned} \tag{3.47}$$

The process for computing this step for each subterm in parallel efficiently on the GPU follows:

1. If the row is marked as a dummy (i.e. the ‘*’ column is 0) then skip to step 7. In such cases, the row can essentially be ignored and its subterm result set immediately to 1 without needing to compute any of the following calculations or steps. (This means the processing of dummy rows essentially amounts to simply doing nothing and waiting for the non-dummy threads to be processed.)
2. Substitute into each subterm expression (i.e. each row), in parallel, the chosen parameter values. This amounts to a simple bitwise multiplication (or AND operation):

$$\begin{array}{c}
 \begin{array}{c|ccccc} 1 & 1 & a & b & c & \cdots \end{array} & \begin{array}{c|ccccc} 1 & a & b & c & \cdots \end{array} \\
 \times \\
 \hline
 \begin{array}{c|ccccc} * & 4\alpha/\pi & \mathbf{a}_\psi & \mathbf{b}_\psi & \mathbf{c}_\psi & \cdots \end{array} & \begin{array}{c|ccccc} & 4\beta/\pi & \mathbf{a}_\phi & \mathbf{b}_\phi & \mathbf{c}_\phi & \cdots \end{array} \\
 \hline
 = \\
 \hline
 \begin{array}{c|ccccc} * & 4\alpha/\pi & \mathbf{a}_\psi a & \mathbf{b}_\psi b & \mathbf{c}_\psi c & \cdots \end{array} & \begin{array}{c|ccccc} & 4\beta/\pi & \mathbf{a}_\phi a & \mathbf{b}_\phi b & \mathbf{c}_\phi c & \cdots \end{array} \\
 \hline
 \end{array}$$

3. Calculate the XOR strings within $\Psi = (\mathbf{a}_\psi a \oplus \mathbf{b}_\psi b \oplus \mathbf{c}_\psi c \oplus \dots)\pi + \alpha$ and $\Phi = (\mathbf{a}_\phi a \oplus \mathbf{b}_\phi b \oplus \mathbf{c}_\phi c \oplus \dots)\pi + \beta$. That is, reduce $(\mathbf{a}_\psi a \oplus \mathbf{b}_\psi b \oplus \mathbf{c}_\psi c \oplus \dots) \rightarrow x$ and $(\mathbf{a}_\phi a \oplus \mathbf{b}_\phi b \oplus \mathbf{c}_\phi c \oplus \dots) \rightarrow y$, where $x, y \in \mathbb{B}$. This can be computed for each row in parallel, relying only on basic arithmetic:

$$\begin{array}{c}
 \begin{array}{c|ccccc} * & 4\alpha/\pi & \mathbf{a}_\psi a & \mathbf{b}_\psi b & \mathbf{c}_\psi c & \cdots \end{array} & \begin{array}{c|ccccc} & 4\beta/\pi & \mathbf{a}_\phi a & \mathbf{b}_\phi b & \mathbf{c}_\phi c & \cdots \end{array} \\
 \downarrow \\
 \hline
 \begin{array}{c|ccccc} * & 4\alpha/\pi & & x & & \end{array} & \begin{array}{c|ccccc} & 4\beta/\pi & & y & & \end{array} \\
 \hline
 \end{array}$$

4. Given $\Psi = x\pi + \alpha \pmod{2\pi}$ and $\Phi = y\pi + \beta \pmod{2\pi}$, calculate $\frac{4\Psi}{\pi} = 4x + \frac{4\alpha}{\pi} \pmod{8}$ and $\frac{4\Phi}{\pi} = 4y + \frac{4\beta}{\pi} \pmod{8}$, such that $\frac{4\Psi}{\pi}, \frac{4\Phi}{\pi} \in \{0, 1, 2, \dots, 7\}$. This too can be calculated for each row in parallel, using only simple operations on integers:

*	$4\alpha/\pi$	x	$4\beta/\pi$	y
\downarrow				
*	$4\Psi/\pi$		$4\Phi/\pi$	

5. The next step is to calculate $e^{i\Psi}$ and $e^{i\Phi}$. However, computing complex exponentials, such as these, is computationally costly. So, to avoid this, a lookup table may be used instead, mapping $4\Psi/\pi \in \{0, 1, 2, \dots, 7\}$ to $e^{i\Psi} \equiv A_\psi + B_\psi\sqrt{2} + i(C_\psi + D_\psi\sqrt{2})$, where this form is used to record the complex numbers as a set of simple fractional numbers³:

$4\psi/\pi$	A_ψ	B_ψ	C_ψ	D_ψ
0	1	0	0	0
1	0	1/2	0	1/2
2	0	0	1	0
3	0	-1/2	0	1/2
4	-1	0	0	0
5	0	-1/2	0	-1/2
6	0	0	-1	0
7	0	1/2	0	-1/2

The same method may be used to compute $e^{i\Phi} \equiv A_\phi + B_\phi\sqrt{2} + i(C_\phi + D_\phi\sqrt{2})$ from $4\Phi/\pi$, and indeed to compute $e^{i(\Psi+\Phi)} \equiv A_{\psi+\phi} + B_{\psi+\phi}\sqrt{2} + i(C_{\psi+\phi} + D_{\psi+\phi}\sqrt{2})$ from $\frac{4\Psi}{\pi} + \frac{4\Phi}{\pi} \bmod 8$. As ever, these calculations can be computed for each row in parallel:

*	$4\Psi/\pi$								$4\Phi/\pi$							
\downarrow																
*	A_ψ	B_ψ	C_ψ	D_ψ	A_ϕ	B_ϕ	C_ϕ	D_ϕ	$A_{\psi+\phi}$	$B_{\psi+\phi}$	$C_{\psi+\phi}$	$D_{\psi+\phi}$				

6. Lastly, having computed $e^{i\Psi}$, $e^{i\Phi}$, and $e^{i(\Psi+\Phi)}$ for each row, ij , each final subterm result may be deduced in parallel by calculating $s_{ij} = 1 + e^{i\Psi} + e^{i\Phi} -$

³More precisely, $A_\psi, B_\psi, C_\psi, D_\psi \in \mathbb{D}$, where \mathbb{D} is the ring of dyadic rational numbers (Giles & Selinger, 2013).

$e^{i(\Psi+\Phi)}$. The result, in each case, may be stored as four simple fractional numbers ($A, B, C, D \in \mathbb{D}$) via the form $s_{ij} \equiv A + B\sqrt{2} + i(C + D\sqrt{2})$. Note that adding two such complex numbers, $e^{i\Psi} + e^{i\Phi}$, in this form amounts to adding the like terms, per lemma 14.

Now, each row will store four simple numbers to record its subterm scalar:

*	A_ψ	B_ψ	C_ψ	D_ψ	A_ϕ	B_ϕ	C_ϕ	D_ϕ	$A_{\psi+\phi}$	$B_{\psi+\phi}$	$C_{\psi+\phi}$	$D_{\psi+\phi}$				
\downarrow																
*	$A_{1,\psi,\phi,-(\psi+\phi)}$				$B_{1,\psi,\phi,-(\psi+\phi)}$				$C_{1,\psi,\phi,-(\psi+\phi)}$				$D_{1,\psi,\phi,-(\psi+\phi)}$			

7. Hiding the subscript labels here for brevity, the culmination of these steps is a matrix wherein each row now records four simple numbers which collectively denote a single subterm value:

A	B	C	D
-----	-----	-----	-----

If the subterm was flagged as a dummy then one may jump straight to this step with $A = 1$, $B = 0$, $C = 0$, $D = 0$.

Lemma 14. *Two complex numbers, $\psi, \phi \in \mathbb{D}[e^{i\pi/4}]$, each expressed via four dyadic rational values, $A_x, B_x, C_x, D_x \in \mathbb{D} \forall x$ may be summed together by adding the like terms:*

$$\begin{aligned}
 & \left[A_\psi + B_\psi\sqrt{2} + i(C_\psi + D_\psi\sqrt{2}) \right] + \left[A_\phi + B_\phi\sqrt{2} + i(C_\phi + D_\phi\sqrt{2}) \right] \\
 &= \left[A_{\psi,\phi} + B_{\psi,\phi}\sqrt{2} + i(C_{\psi,\phi} + D_{\psi,\phi}\sqrt{2}) \right] \tag{3.48}
 \end{aligned}$$

where here:

$$\begin{aligned}
 A_{\psi,\phi} &= A_{\psi} + A_{\phi} \\
 B_{\psi,\phi} &= B_{\psi} + B_{\phi} \\
 C_{\psi,\phi} &= C_{\psi} + C_{\phi} \\
 D_{\psi,\phi} &= D_{\psi} + D_{\phi}
 \end{aligned} \tag{3.49}$$

Taking the parameterised scalar matrix of example 4 and evaluating it for $(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}) = (0, 1, 1, 0)$, the steps would proceed as follows, with each row being processed in parallel:

*	4α	\mathfrak{a}_{ψ}	\mathfrak{b}_{ψ}	\mathfrak{c}_{ψ}	\mathfrak{d}_{ψ}	4β	\mathfrak{a}_{ϕ}	\mathfrak{b}_{ϕ}	\mathfrak{c}_{ϕ}	\mathfrak{d}_{ϕ}
1	2	1	1	0	0	7	0	1	1	1
1	4	0	0	1	0	3	1	0	0	1
1	7	1	1	1	1	6	1	0	1	1
1	6	0	1	1	0	3	1	1	0	1
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	2	1	0	0	0
1	2	0	1	1	0	1	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0

↓ step 2 ↓

$*$	4α	$0\mathfrak{a}_\psi$	$1\mathfrak{b}_\psi$	$1\mathfrak{c}_\psi$	$0\mathfrak{d}_\psi$	4β	$0\mathfrak{a}_\phi$	$1\mathfrak{b}_\phi$	$1\mathfrak{c}_\phi$	$0\mathfrak{d}_\phi$
1	2	0	1	0	0	7	0	1	1	0
1	4	0	0	1	0	3	0	0	0	0
1	7	0	1	1	0	6	0	0	1	0
1	6	0	1	1	0	3	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	2	0	0	0	0
1	2	0	1	1	0	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0

↓ step 3 ↓

$*$	4α	x	4β	y
1	2	1	7	0
1	4	1	3	0
1	7	0	6	1
1	6	0	3	1
0	0	0	0	0
0	0	0	0	0
1	0	1	2	0
1	2	0	1	1
0	0	0	0	0

↓ step 4 ↓

$*$	$4\Psi/\pi$	$4\Phi/\pi$
1	6	7
1	0	3
1	7	2
1	6	7
0	0	0
0	0	0
1	4	2
1	2	5
0	0	0

↓ step 5 ↓

$*$	A_ψ	B_ψ	C_ψ	D_ψ	A_ϕ	B_ϕ	C_ϕ	D_ϕ	$A_{\psi+\phi}$	$B_{\psi+\phi}$	$C_{\psi+\phi}$	$D_{\psi+\phi}$
1	0	0	-1	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$
1	1	0	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	$\frac{1}{2}$
1	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	0	1	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$
1	0	0	-1	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$
0	—	—	—	—	—	—	—	—	—	—	—	—
0	—	—	—	—	—	—	—	—	—	—	—	—
1	-1	0	0	0	0	0	1	0	0	0	-1	0
1	0	0	1	0	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	$-\frac{1}{2}$
0	—	—	—	—	—	—	—	—	—	—	—	—

↓ step 6 ↓

A	B	C	D
1	1	-1	0
2	0	0	0
1	0	1	-1
1	1	-1	0
1	0	0	0
1	0	0	0
0	0	2	0
1	-1	1	0
1	0	0	0

This result is consistent with that of equation 3.46, albeit with the constant factors, C_1, C_2, C_3 , neglected for now (and some redundant factors of 1 included). The significance of this approach is that it processes each subterm in a parallel thread, while relying only on simple arithmetic and performing the same calculations at each step across these threads. It hence makes efficient use of GPU parallelism that would not be achievable without such considerations as presented.

A CUDA kernel that implements this procedure is shown in listing 3.1⁴. This code is hopefully self-explanatory, though note that the variable names may not be consistent with what is used above.

Listing 3.1: A CUDA kernel for GPU-parallelised evaluation of parameterised subterms.

```
__global__ void calc_subterms(int n, char* rowGenuine, char*
    rowConstA, char* rowConstB, char* rowsumA, char* rowsumB,
```

⁴Note that this implementation stores scalars in the form $A + iB$ with $A, B \in \mathbb{R}$, rather than $A + B\sqrt{2} + i(C + D\sqrt{2})$ with $A, B, C, D \in \mathbb{D}$. This marginally reduces precision but improves runtime.

```
double* real, double* imag) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i > n - 1) return;

    if (rowGenuine[i] == 0) { // ignore dummy rows
        real[i] = 1.0;
        imag[i] = 0.0;
        return;
    }

    const double root2 = 1.414213562373095;
    char a = (rowConstA[i] + (rowsumA[i] * 4)) % 8;
    char b = (rowConstB[i] + (rowsumB[i] * 4)) % 8;
    char c = (a + b) % 8;

    double factor_re = 1.0, factor_im = 0.0;

    // 1 + cexp(a) + cexp(b) - cexp(c)...

    switch (a) { // cexp(a)
    case 0: factor_re += 1.0; break;
    case 1: factor_re += 0.5*root2; factor_im += 0.5 * root2;
        break;
    case 2: factor_im += 1.0; break;
    case 3: factor_re += -0.5*root2; factor_im += 0.5*root2;
        break;
    case 4: factor_re += -1.0; break;
    case 5: factor_re += -0.5*root2; factor_im += -0.5*root2;
        break;
    case 6: factor_im += -1.0; break;
    case 7: factor_re += 0.5*root2; factor_im += -0.5*root2;
        break;
    }
}
```

```
switch (b) { // cexp(b)
case 0: factor_re += 1.0; break;
case 1: factor_re += 0.5*root2; factor_im += 0.5 * root2;
      break;
case 2: factor_im += 1.0; break;
case 3: factor_re += -0.5*root2; factor_im += 0.5*root2;
      break;
case 4: factor_re += -1.0; break;
case 5: factor_re += -0.5*root2; factor_im += -0.5*root2;
      break;
case 6: factor_im += -1.0; break;
case 7: factor_re += 0.5*root2; factor_im += -0.5*root2;
      break;
}

switch (c) { // cexp(c)
case 0: factor_re -= 1.0; break;
case 1: factor_re -= 0.5*root2; factor_im -= 0.5 * root2;
      break;
case 2: factor_im -= 1.0; break;
case 3: factor_re -= -0.5*root2; factor_im -= 0.5*root2;
      break;
case 4: factor_re -= -1.0; break;
case 5: factor_re -= -0.5*root2; factor_im -= -0.5*root2;
      break;
case 6: factor_im -= -1.0; break;
case 7: factor_re -= 0.5*root2; factor_im -= -0.5*root2;
      break;
}

real[i] = factor_re;
imag[i] = factor_im;
```

```

    return;
}

```

3.3.4 Computing the Terms

Having demonstrated how the subterms of a parameterised scalar expression may be evaluated in an efficient and parallelised manner on a GPU, it remains to multiply together all the subterms with each term, before finally summing together these terms.

The subterms within each term may be multiplied together sequentially, making use of lemma 15. With each term assigned to its own thread, this too may be parallelised. So, instead of each row corresponding to a parallel thread, now each block of n rows is processed on a single thread.

With an unlimited number of threads, where each term could truly be computed in parallel, this would require $O(n)$ steps, where n is the number of subterms per term. More practically, with a finite $t < m$ threads (where m is the number of terms), this would require $\frac{m}{t}$ sequentially processed batches of t parallel threads. Hence, the runtime complexity of this step may be given as follows:

- If $t \geq m$, the time complexity is $O(n)$.
- If $t < m$, the time complexity is $O(\frac{nm}{t})$.

Lemma 15. *Two complex numbers, $\psi, \phi \in \mathbb{D}[e^{i\pi/4}]$, each expressed via four dyadic rational values, $A_x, B_x, C_x, D_x \in \mathbb{D} \forall x$, in the form:*

$$\begin{aligned}\psi &= A_\psi + B_\psi\sqrt{2} + i(C_\psi + D_\psi\sqrt{2}) \\ \phi &= A_\phi + B_\phi\sqrt{2} + i(C_\phi + D_\phi\sqrt{2})\end{aligned}\tag{3.50}$$

may be multiplied together, $\vartheta = \psi \times \phi$, using only simple real arithmetic:

$$\vartheta = A_{\vartheta} + B_{\vartheta}\sqrt{2} + i \left(C_{\vartheta} + D_{\vartheta}\sqrt{2} \right) \quad (3.51)$$

where:

$$\begin{aligned} A_{\vartheta} &= A_{\psi}A_{\phi} + 2B_{\psi}B_{\phi} - C_{\psi}C_{\phi} - 2D_{\psi}D_{\phi} \\ B_{\vartheta} &= A_{\psi}B_{\phi} + B_{\psi}A_{\phi} - C_{\psi}D_{\phi} - D_{\psi}C_{\phi} \\ C_{\vartheta} &= A_{\psi}C_{\phi} + 2B_{\psi}D_{\phi} + C_{\psi}A_{\phi} + 2D_{\psi}B_{\phi} \\ D_{\vartheta} &= A_{\psi}D_{\phi} + B_{\psi}C_{\phi} + C_{\psi}B_{\phi} + D_{\psi}A_{\phi} \end{aligned} \quad (3.52)$$

Making use again of lemma 15, the scalar factors, $C_i \in \mathbb{C}$ (or, more precisely, $C_i \in \mathbb{D}[e^{i\pi/4}]$), of each term, i , may also be multiplied in as appropriate.

Example 6. *Returning to the previous example, this step would reduce the matrix as follows:*

A	B	C	D
1	1	-1	0
2	0	0	0
1	0	1	-1
1	1	-1	0
1	0	0	0
1	0	0	0
0	0	2	0
1	-1	1	0
1	0	0	0

\downarrow multiply together subterms \downarrow

A	B	C	D
4	0	-4	0
1	1	-1	0
-2	0	2	-2

\downarrow multiply by constant factor, $C_i \downarrow$

A	B	C	D
$\frac{1}{8}$	0	$-\frac{1}{8}$	0
$\frac{3}{16}$	$\frac{1}{8}$	$-\frac{1}{16}$	$-\frac{3}{16}$
$-\frac{1}{2}$	0	$\frac{1}{2}$	$-\frac{1}{2}$

This step is equivalent to reducing equation 3.46 to:

$$g(0, 1, 1, 0) = C_1(4 - 4i) + C_2(1 + \sqrt{2} - i) + C_3(-2 + 2i - 2i\sqrt{2}) \quad (3.53)$$

and lastly:

$$\begin{aligned}
 g(0, 1, 1, 0) &= \left(\frac{1}{32}\right)(4 - 4i) + \left(\frac{\sqrt{2}}{8} - \frac{1}{16}i\right)(1 + \sqrt{2} - i) \\
 &\quad + \left(\frac{1}{4}\right)(-2 + 2i - 2i\sqrt{2}) \\
 &= \left(\frac{1}{8} - \frac{1}{8}i\right) + \left(\frac{3}{16} + \frac{\sqrt{2}}{8} - \frac{1}{16}i - \frac{3\sqrt{2}}{16}i\right) \\
 &\quad + \left(-\frac{1}{2} + \frac{1}{2}i - \frac{\sqrt{2}}{2}i\right)
 \end{aligned} \quad (3.54)$$

3.3.5 Parallelised Summation Algorithm

Now each term has been reduced to a complex number, expressed with four real numbers, the final step is to sum these terms together. Via sequential (i.e. single core CPU) computation, there is no more efficient method for summing all the elements of a length n array of numbers than to simply iterate through them and maintain a tally (Cormen, Leiserson, Rivest, & Stein, 2022). This, of course, offers a runtime complexity of $O(n)$.

However, in allowing parallel processing (such as that enabled by a GPU), it is possible to improve upon this with methods (JáJá, 1992) that (asymptotically and given enough parallel threads) achieve $O(\log n)$. Algorithm 3 describes such a method, where `PSum` is a GPU kernel executing on n parallel threads (each with a successive unique index provided by `GetThreadIdx()`).

Algorithm 3 The GPU-parallelised summation algorithm

```

1: Initialise and populate  $ARR[N_{ELEMS}]$ 
2:
3: procedure PSUM( $split, gap$ ) $\ll n \gg$ 
4:    $i \leftarrow \text{GETTHREADINDEX}()$ 
5:    $elem \leftarrow i \times split$ 
6:   if  $elem + gap < N_{ELEMS} - 1$  then
7:      $ARR[elem] \leftarrow ARR[elem] + ARR[elem + gap]$ 
8:   end if
9: end procedure
10:
11:  $split \leftarrow 2$ 
12:  $gap \leftarrow 1$ 
13: while  $gap < N_{ELEMS}$  do
14:    $n_{chunks} \leftarrow \lceil N_{ELEMS}/split \rceil$ 
15:   PSUM( $split, gap$ ) $\ll n_{chunks} \gg$ 
16:    $split \leftarrow split \times 2$ 
17:    $gap \leftarrow gap \times 2$ 
18: end while

```

Best understood by demonstration, this procedure is illustrated in example 7.

Example 7. *As an example, consider the following 10-element array of integers:*

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Every adjacent pair of elements may be allocated a single GPU thread. For a general n -element array, this means allocating $n/2$ threads, which in this example corresponds to 5 threads, divided as follows:

$$[0, 1, \mid 2, 3, \mid 4, 5, \mid 6, 7, \mid 8, 9]$$

Each thread, running in parallel, may then sum the two elements under its consideration and overwrite its left element with the result, like so:

$$[1, 1, \mid 5, 3, \mid 9, 5, \mid 13, 7, \mid 17, 9]$$

The right element of each pair can hereafter be ignored and so is written in grey here. The process may then repeat, with $n/4$ threads being allocated to consider adjacent 4-elements groups, like so:

$$[1, 1, 5, 3, \mid 9, 5, 13, 7, \mid 17, 9]$$

Now, the leftmost element of each thread will be incremented by the value contained 2 elements to its right (unless that index exceeds the length of the array):

$$[6, 1, 5, 3, \mid 22, 5, 13, 7, \mid 17, 9]$$

This process may then repeat iteratively, halving the number of threads needed at each step and doubling the number of elements considered by each thread and the gap between their relevant (non-grey) pairs of elements. The example case would proceed as follows:

$$[6, 1, 5, 3, 22, 5, 13, 7, \mid 17, 9]$$

$$[28, 1, 5, 3, 22, 5, 13, 7, \mid 17, 9]$$

$$[28, 1, 5, 3, 22, 5, 13, 7, 17, 9]$$

[45, 1, 5, 3, 22, 5, 13, 7, 17, 9]

Once the number of required threads is reduced to 1 (itself containing the whole array), the procedure is complete and the final result (in this case 45) — being the sum of all elements in the original array — may be found in the leftmost element.

This method can equivalently be applied to multiply, rather than sum, all elements of an array, and in either case requires $\lceil \log_2 n \rceil$ iterations, rather than n as sequential processing would necessitate. This result is in fact a theoretical limit, however, as it assumes an unlimited number of available parallel threads. In fact, if the number of available threads is fewer than $n/2$, then full parallelisation is not achieved and the true number of iterations increases beyond the theoretical minimum of $\lceil \log_2 n \rceil$. Hence, a more practical expression of the time complexity, which recognises a finite t threads is:

- If $t \geq \frac{n}{2}$, the time complexity is $O(\log n)$.
- If $t < \frac{n}{2}$, the time complexity is $O\left(\frac{n}{t} \log n\right)$.

Therefore, for non-trivial n , an improvement the parallelised array summation/-multiplication method outperforms the sequential method when $t > \log_2 n$. In reality, a typical GPU may have several hundreds, or even thousands, of parallel threads (Martineau, Atkinson, & McIntosh-Smith, 2018), meaning that in any feasible case:

$$t \gg \log_2 n \quad (3.55)$$

and hence this method almost invariably results in drastic runtime improvements versus the sequential alternative. Furthermore, data pipelining (as described in section 2.5.5) may be employed when sending the initial n -element array from the CPU to the GPU, and the result to be returned to the CPU is simply a single numerical value. Consequently, for non-trivial cases the data transfer time of this

method turns out to be negligible.

With the evaluated scalar terms already expressed in a simple form as rows of a matrix on the GPU, algorithm 3 may be utilised, together with lemma 14, to efficiently sum these terms together. This makes one final use of GPU parallelism and produces a single complex number as a result, which may be returned to the CPU in negligible time.

This returned result gives the final result of evaluating the original parameterised scalar expression for a particular set of parameter values.

Example 8. *Returning one final time to example 6, this step will reduce the term data as follows:*

A	B	C	D
$\frac{1}{8}$	0	$-\frac{1}{8}$	0
$\frac{3}{16}$	$\frac{1}{8}$	$-\frac{1}{16}$	$-\frac{3}{16}$
$-\frac{1}{2}$	0	$\frac{1}{2}$	$-\frac{1}{2}$
\downarrow Sum the terms (i.e. rows) \downarrow			
A	B	C	D
$-\frac{3}{16}$	$\frac{1}{8}$	$\frac{5}{16}$	$-\frac{11}{16}$

This final result, equivalent to reducing equation 3.54 to:

$$g(0, 1, 1, 0) = -\frac{3}{16} + \frac{\sqrt{2}}{8} + \frac{5}{16}i - \frac{11\sqrt{2}}{16}i \quad (3.56)$$

may then be returned to the CPU.

3.3.6 Summary of the New Method

There is much utility in taking many repeat evaluations of a parameterised ZX-diagram, for various sets of parameter values. Such use cases include, as will be seen, parameterised stabiliser decompositions and mid-circuit measurements. Of most relevance to this thesis, however, is the case of classical simulation, with parameters arising from open circuit outputs.

Conventionally, the ZX-calculus approach to this problem is that of (Kissinger & van de Wetering, 2022), described in section 2.3. For each evaluation, this essentially involves substituting parameter values into a parameterised Clifford+T ZX-diagram and decomposing it into many Clifford terms, which may be reduced and summed to arrive at the final scalar result. This whole process need then be repeated for every subsequent evaluation, with a new set of parameter values.

As seen in section 2.3, decomposing a t T-count ZX-diagram results in a sum of $2^{\alpha t}$ stabiliser terms, with α determined by the decomposition(s) used. For this reason, decomposing a ZX-diagram of even a modest T-count is a very slow process. Hence, repeating this slow process for many independent evaluations increases this slow runtime manyfold.

The approach described in this chapter seeks to address this issue. Firstly, by generalising the decomposition and simplification strategy of ZX-calculus to support (reasonably restricted) parameterised phases, this enables this slow process to be computed just for all possible parameter values. The result is a parameterised scalar expression which may be evaluated many times for various sets of parameter values, without needing to repeat any slow ZX-calculus reduction. Lastly, given that such an evaluation of a parameterised scalar requires only very simple, and highly-parallelisable, arithmetic, this step can be performed very efficiently

and rapidly, making use of GPU hardware.

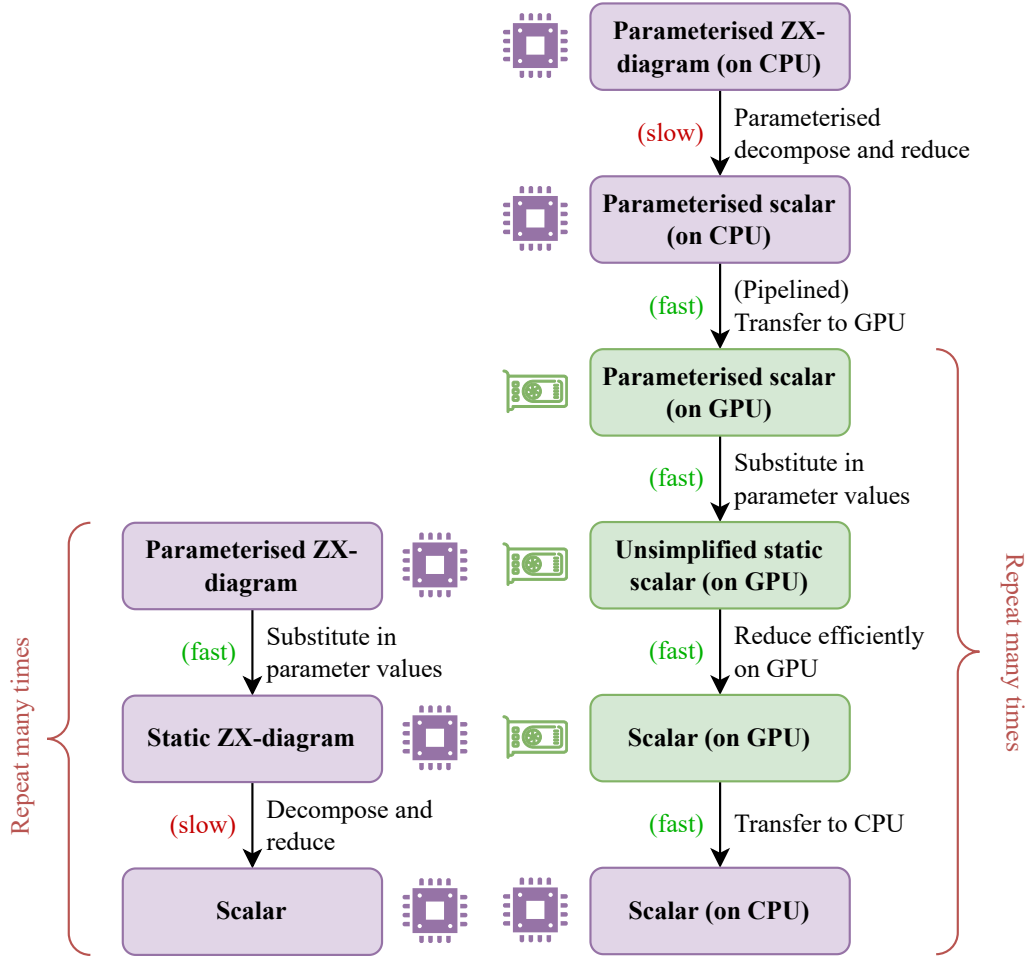
Hence, in summary, with this approach one need only compute the slow ZX-calculus reduction step once, with every subsequent evaluation (for a new set of parameter values) being very quick to calculate. This is as opposed to repeating the whole slow ZX-calculus reduction from scratch for every new evaluation. Figure 3.5 illustrates this distinction with an overview of the steps involved in each method.

With the method outlined in this chapter, every successive evaluation of a non-trivial parameterised Clifford+T ZX-diagram is expected to be calculable much more rapidly than via the conventional approach of (Kissinger & van de Wetering, 2022). However, this method still requires one instance of ZX-calculus decomposition and reduction to transform the initial parameterised ZX-diagram into a GPU-ready parameterised scalar expression. Consequently, if one were taking just a single evaluation, then no improvement would be expected, but as one takes an ever greater number of evaluations this initialisation time begins to become negligible and the overall runtime difference between the methods much more drastic. Lemmas 16 and 17 provide a stronger mathematical analysis of this point.

Lemma 16. *Let T_{eval}^{old} be average time required to calculate a single evaluation of a given parameterised ZX-diagram with the conventional ‘old’ method of (Kissinger & van de Wetering, 2022). The overall time required to calculate N such evaluations with this method is then:*

$$T_{total}^{old} = NT_{eval}^{old} \quad (3.57)$$

Alternatively, the ‘new’ method presented in this chapter requires a one-off initialisation time, T_{init}^{new} , with subsequent evaluations computable in an average time T_{eval}^{new} . This method thus requires the following runtime to calculate N evaluations



(a) The conventional CPU-based approach, introduced in (Kissinger & van de Wetering, 2022).

(b) The parallelised GPU-based approach outlined in this chapter.

Figure 3.5: A comparison of the two procedures for repeated evaluation of a parameterised ZX-diagram for various sets of parameter values. The boxes show the state of the data at each step, connected by arrows indicating the processes that update these data, together with a qualitative note of the speed of each process. Purple boxes represent data being on the CPU (CPU icon) and green boxes data on the GPU (GPU icon).

of the given parameterised ZX-diagram:

$$T_{total}^{new} = T_{init}^{new} + NT_{eval}^{new} \quad (3.58)$$

The **speedup factor**, S_N , achieved by using the new method versus the old for taking N evaluations of a particular parameterised ZX-diagram is then given by:

$$S_N = \frac{T_{total}^{old}}{T_{total}^{new}} \quad (3.59)$$

As the initialisation step of the new method and an evaluation via the old method are both instances of a full decomposition of the given ZX-diagram (albeit with the former maintaining parametric phases), it follows that, within a small factor:

$$T_{init}^{new} \sim T_{eval}^{old} \quad (3.60)$$

As a result, for small N , the new method is not expected to offer a meaningful runtime reduction, but as N increases the runtime ratio of the two methods becomes increasingly drastic:

$$\frac{T_{total}^{old}}{T_{total}^{new}} \rightarrow \frac{T_{eval}^{old}}{T_{eval}^{new}} \quad \text{as } N \rightarrow \infty \quad (3.61)$$

This $N \rightarrow \infty$ result may be labelled the **terminal speedup factor**, S_∞ :

$$S_\infty = \frac{T_{eval}^{old}}{T_{eval}^{new}} \quad (3.62)$$

Lemma 17. *The relationship between the speedup factor, S_N , and the number of evaluations, N , can be visualised as a sigmoid curve of the form:*

$$S_N(N) = S_\infty \cdot \frac{N}{N_{inflec} + N} \quad (3.63)$$

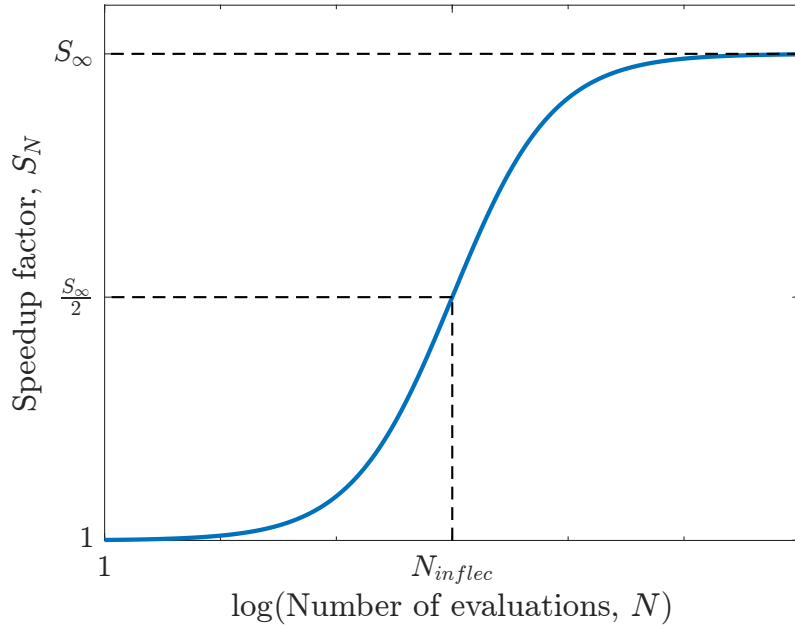


Figure 3.6: The relationship between the speedup factor, S_N , and the number of evaluations, N , for any particular ZX-diagram may be visualised as a sigmoid curve. Note the logarithmic x -axis.

as in figure 3.6, where:

$$N_{inflec} = \frac{T_{init}^{new}}{T_{eval}^{new}} \quad (3.64)$$

is the inflection point: that is the value of N at which $S_N = S_\infty/2$.

Proof. Using lemma 16:

$$S_N = \frac{T_{total}^{old}}{T_{total}^{new}} = \frac{NT_{eval}^{old}}{T_{init}^{new} + NT_{eval}^{new}} = \frac{T_{eval}^{old}}{T_{eval}^{new}} \cdot \frac{N}{N_{inflec} + N} = S_\infty \cdot \frac{N}{N_{inflec} + N} \quad (3.65)$$

□

3.4 Application to Classical Simulation

The chapter thus far has presented a method by which parameterised ZX-diagrams may be reduced to parameterised scalars which in turn may be repeatedly evaluated for various sets of parameter values very rapidly, making use of GPU parallelism. As such, it is applicable whenever one wishes to reduce and evaluate every static ZX-diagram belonging to a particular parametrically symmetric set. This is, in fact, a fairly common scenario, occurring in some stabiliser decompositions, circuits with measurements, and, of course, classical simulation.

3.4.1 Repeated Strong Simulation

As discussed in section 2.3, a Clifford+T circuit may be strongly simulated for a particular measurement outcome by ‘plugging’ its outputs with the appropriate bitstring, before reducing it to a scalar via the strategies of ZX-calculus. If one desired to strongly simulate such a circuit many times to, for instance, measure the probability of various measurement outcomes, then it is perhaps obvious how the methods of this chapter could be of aid. Simply plugging the outputs of the circuit with a parameterised bitstring would enable it to be reduced to a parameterised scalar, whereupon the amplitudes corresponding to various outcome measurements could be calculated very hastily using this chapter’s approach.

3.4.2 Computing Individual Marginal Probabilities

Section 2.3 also demonstrated how a Clifford+T ZX-diagram may be *weakly* simulated by computing its marginal probabilities through the ‘*summing*’ or ‘*doubling*’ method. The techniques of this chapter may be applied to either method, in slightly distinct ways.

The summing method is simply to strongly simulate every element in a parametrically symmetric set of scalar ZX-diagrams. This is exactly the situation described above, and the relevance of this chapter here is apparent. The only additional step required would be to sum the squares of the resulting evaluations (which could be done ‘on the fly’ as they are each returned to the CPU).

Applied in this way, every individual instance of computing a marginal probability may be sped up by effectively parallelising the sum in this ‘*sum of squares*’ approach. This can hence speed up the calculation of a single instance of weak simulation.

3.4.3 Repeated Weak Simulation

Alternatively, there are many cases in which the doubling method of computing marginal probabilities is more appropriate and efficient than the summing method. In such cases, the techniques of this chapter may again be applied, provided one wishes to compute *many* instances of weak simulation, rather than just one.

In this situation, each $(1 \leq k \leq n)$ marginal probability computation of an n -qubit circuit, U , can be computed parametrically:

$$P(\mathbf{a}_1 \cdots \mathbf{a}_k) = \text{Diagram} \quad (3.66)$$

With this approach, at each iteration (i.e. in incrementing k) of computing the next marginal probability, one may deduce the next bit in the final bitstring, for any number N of independent bitstrings. Hence, one will ultimately produce N such bitstrings denoting N independent samples of weak simulation, while only

ever reducing n doubled ZX-diagram, given n qubits.

This is summarised in algorithm 4, where:

$$P(\mathbf{a}_1 \cdots \mathbf{a}_k) \stackrel{r}{\leftarrow} \langle 0 \cdots 0 | U^\dagger (|\mathbf{a}_1 \cdots \mathbf{a}_k\rangle \langle \mathbf{a}_1 \cdots \mathbf{a}_k| \otimes I_{n-k}) U | 0 \cdots 0 \rangle \quad (3.67)$$

denotes the reduction of the doubled marginal probability ZX-diagram (equation 3.66) to a parameterised scalar expression, $P(\mathbf{a}_1 \cdots \mathbf{a}_k)$, denoting its outcome probability. $P(B)$ then denotes a GPU-based evaluation of this scalar expression for the bitstring $B = a_1 \cdots a_k$, where $a_i \in \mathbb{B} \ \forall i$. Meanwhile, $X||Y$ denotes the concatenation of X and Y , such that $[a, b, c]||d = [a, b, c, d]$, and $\text{Rand}()$ returns a random floating point number in the range $[0, 1)$.

Algorithm 4 Algorithm for efficient repeated weak simulation

```

1: Input: A quantum circuit  $U$  with  $n_{\text{qubits}}$  qubits, and  $N_{\text{evals}}$ 
2:
3:  $B_i = [] \ \forall i \in \{1, \dots, n_{\text{qubits}}\}$  ▷ Initialise empty lists
4: for  $k = 1$  to  $n_{\text{qubits}}$  do
5:    $P(\mathbf{a}_1 \cdots \mathbf{a}_k) \stackrel{r}{\leftarrow} \langle 0 \cdots 0 | U^\dagger (|\mathbf{a}_1 \cdots \mathbf{a}_k\rangle \langle \mathbf{a}_1 \cdots \mathbf{a}_k| \otimes I_{n-k}) U | 0 \cdots 0 \rangle$ 
6:   for  $i = 1$  to  $N_{\text{evals}}$  do
7:      $P_{B_i||0} = P(B_i||0)$  ▷ GPU-based evaluation
8:      $b \leftarrow 0$  if  $\text{Rand}() < P_{B_i||0}$  else  $1$ 
9:      $B_i = B_i||b$ 
10:  end for
11: end for
12:
13: Output:  $B_i \ \forall i$ 

```

There are further optimisations that may be made to this algorithm, with some particular redundancy among the first few iterations of k , where identical evaluations are inevitable. But such considerations are perhaps superfluous.

3.5 Results

3.5.1 Experimental Setup

The method detailed in this chapter was benchmarked against the conventional, CPU-based and non-parameterised, method of (Kissinger & van de Wetering, 2022) for fully reducing and evaluating randomly generated Clifford+T circuits. These circuits were generated using the existing *generate.cliffordT* function of *Pyzx*⁵ (Kissinger & van de Wetering, 2018, 2020a), before being translated into ZX-diagrams. A wide range of T-counts were considered, as a means to vary the size and complexity of the initial circuits, with the parameter count (or *P-count*) in each case fixed at 10.

Specifically, the new method is compared against the *Quizx* (Kissinger & van de Wetering, 2021) implementation of the conventional method, which is the Rust port of *Pyzx*, developed to maximise speed. As such, direct comparisons of the runtimes of the respective methods are reasonable and fair (certainly within acceptable error margins), particularly as Rust and C are languages comparable in speed (Y. Zhang, Zhang, Portokalidis, & Xu, 2022).

Furthermore, *Quizx* utilises some degree of *CPU* parallelism which contributes to its high speeds, and so runtime comparisons against it may be considered understated. Given this, the speedup rates quoted among the upcoming results may be seen as conservative measurements, and would be expected to be even greater if compared to a single-core CPU implementation of (Kissinger & van de Wetering, 2022) in *Quizx*.

Lastly, note that all experimental measurements among these results were pro-

⁵*Pyzx* is a Python package which implements ZX-calculus, equipped with functions to visual, manipulate, and simplify ZX-diagrams.

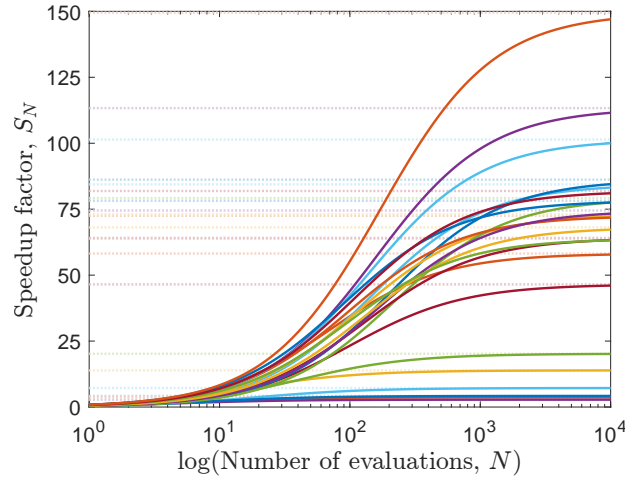


Figure 3.7: The measured speedup factors versus the number of evaluations, for randomly generated parameterised circuits of various T-counts. The faint dotted lines indicate the corresponding terminal speedup factors, which are approached asymptotically.

duced on commercial hardware containing a *6-core 2.69GHz Intel i5-11400H CPU* and an *8GB NVIDIA GeForce GTX 1650 GPU*, plus *8GB SODIMM RAM*.

3.5.2 Experimental Measurements

In each experiment, the measured speedup factors varied with the number of evaluations in a manner consistent with that predicted by lemma 17. Figure 3.7 illustrates this for a random subset of the experiments (selecting one experiment for each unique T-count).

Evidently, there is significant variation among these results, with a terminal speedup factor ranging from < 1 to 149.5. This, as will soon be seen, is due largely (though not entirely) to variation among the initial T-counts of these circuits. Nevertheless, in all these cases, several thousand evaluations prove sufficient to reach over

90% of the terminal speedup factor:

$$S_N > 0.9S_\infty \quad \text{when } N \gtrsim 10^3 \quad (3.68)$$

To consider a specific typical example, the conventional Quizx approach was able to evaluate a particular 20-qubit circuit of T-count 36 and P-count 5 in 472 ± 112 ms. On the other hand, the parameterised GPU-based approach, after an initialisation time of 651 ± 116 ms, was able to evaluate the same circuit in just 4.66 ± 0.17 ms. This produces a terminal speedup factor $S_\infty = 472/4.66 \approx 101$. This means, given enough evaluations, the new method will perform up to 101 times more quickly than the old. In this case, a runtime improvement is observed for any non-trivial number of evaluations, with 100 evaluations being enough to see a speedup factor of 42 and 1,000 enough to see a speedup factor of 89.

From lemmas 16 and 17, one may note that there are two important metrics which may be used to quantify the effectiveness of this method, as compared against the conventional approach (Kissinger & van de Wetering, 2022), for any given parameterised ZX-diagram. These metrics are:

- the **terminal speedup factor**, S_∞ , which measures the theoretical maximum runtime improvement for the given parameterised ZX-diagram, and
- the **inflection point**, N_{inflex} , which measures how many evaluations are required for the speedup factor to reach half of its theoretical maximum, and which effectively quantifies the rate at which the speedup factor increases against the number of evaluations taken.

For each experiment, these two metrics describe and quantify the behaviour of the corresponding sigmoid curve (such as those of figure 3.7). Figure 3.8 plots these metrics against the initial (post Clifford simplification) T-counts of the parame-

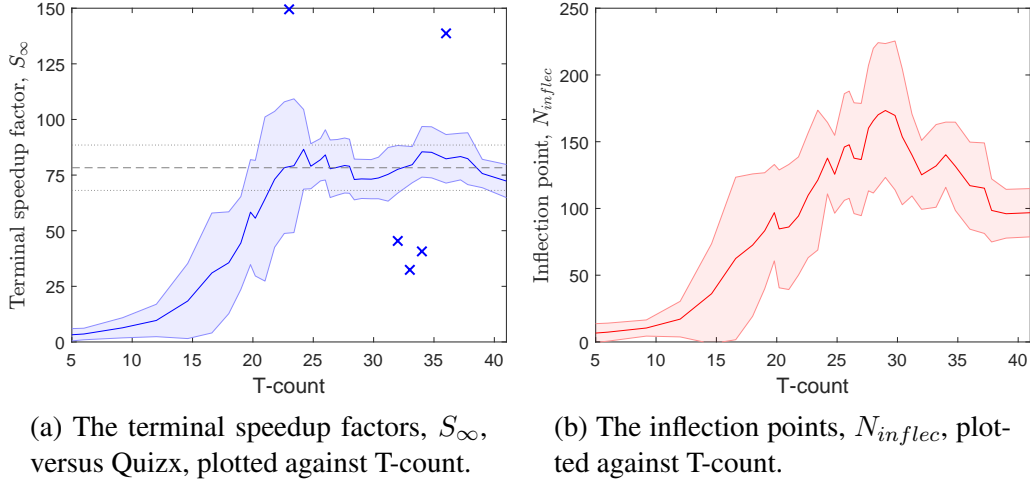


Figure 3.8: The measured (a) terminal speedup factors and (b) inflection points of various randomly generated parameterised ZX-diagrams versus the conventional Quizx implementation (Kissinger & van de Wetering, 2022), plotted against the initial T-counts (after Clifford simplification). These plots show a (5-point window) moving average of the measured results, with error margins given by the standard deviation. Extreme outlier measurements have been excluded and indicated with an ‘x’, and figure (a) includes a dashed grey line indicating the average S_∞ (with standard deviation error margins) across all (non-outlier) measurements of T-counts ≥ 25 , where S_∞ appears to plateau.

terised ZX-diagrams to emphasise how they relate to the size and complexity of the circuits.

From figure 3.8a, it is apparent that for very small T-counts, this method offers minimal speedup versus Quizx. This is to be expected, as ZX-diagrams of small T-counts decompose into a small number of stabiliser terms — too few to take full advantage of GPU parallelism. This is especially true when the number of terms is fewer than the number of available parallel threads. However, as can be observed, when the T-count is increased beyond trivial scales, the effectiveness of the method is likewise increased, and drastically so, with terminal speedup factors near 100 becoming commonplace.

While the terminal speedup factor initially correlates positively with the T-count,

it evidently plateaus when the T-count extends beyond around 25. This is believed to be the scale beyond which such ZX-diagrams reduce into a number of terms sufficient to occupy the available parallel threads (on the specified GPU) at full capacity. The precise point at which this plateau occurs, therefore, is highly dependent upon the specific hardware used, with a greater number of GPU threads equating to a later plateau. (Indeed, given a theoretical infinite number of parallel threads, one would expect no such plateau, with the terminal speedup factor continuing to grow endlessly with the T-count.) In the results presented here, the average terminal speedup factor across all experiments (excluding extreme outliers) of T-counts ≥ 25 is 78.3 ± 10.2 .

Within the plateau region, most cases produced terminal speedup factor measurements between approximately 60 and 100. However, there were a handful of exceptional cases (shown in figure 3.8a as outliers), with S_∞ as low as 32.4 and as high as 149.5. While such cases are less common, they demonstrate some interesting phenomena. The effectiveness of this method on a particular parameterised ZX-diagram depends on how well it is able to reduce while maintaining parametric generality. This can vary significantly from one ZX-diagram to another, even for those of an equivalent T-count.

As per lemma 8, owing largely to the problematic π -commutation rule, reducing a parameterised Clifford+T ZX-diagram results in more terms than reducing its static equivalent. In practice, the difference in the number of terms appears to be as random as the circuits themselves and independent of the T-count. Among the experiments conducted in this chapter, the number of terms attained from reducing the parameterised Clifford+T ZX-diagrams was, on average, 1.71 times greater than those numbers attained from reducing their static counterparts. This is accompanied by a standard deviation of 0.44 and a range of [1.12, 3.29].

Given the consistency of this ratio, it follows that the numbers of terms produced for a given ZX-diagram by the two methods are approximately proportional (\propto). Given also that the runtime of reducing a ZX-diagram is proportional to the number of stabiliser terms to which it reduces, it stands to reason that $T_{eval}^{old} \propto T_{init}^{new}$, and therefore $S_{\infty} \propto N_{inflex}$. This explains why figures 3.8a and 3.8b are approximately proportional with a proportionality constant lingering relatively near the expected 1.71.

In the rare extreme cases, such as when this ratio was measured as 3.29, the parameterised method has considerably more terms to compute compared to the Quizx method and hence the terminal speedup factor falls short of the norm. This phenomenon is what gives rise to the lower outliers of figure 3.8a. Such extreme cases arise from particularly unfortunate ZX-diagrams which are poor at simplifying while maintaining generality. This usually occurs if the π -commutation roadblocks discussed in section 3.2.3 are encountered very earlier on, before much decomposition has yet occurred, such as during the initial Clifford simplification stage.

The higher (fortunate) outliers, meanwhile, likely benefit from encountering very few, if any, instances of such roadblocks (and hence require both methods to compute a near-equivalent number of terms). More significantly, they likely arise from parameterised ZX-diagrams which happen to reduce very neatly into parameterised scalar expressions wherein each term contains a relatively small number of subterms. In such fortunate cases, this would ease the computational workload of the GPU-based method quite drastically.

In general, however, the number of subterms per term was found to be fairly consistent among the experiments, with relatively minimal fluctuation, even across T-counts. This helps to explain the relatively narrow error margins seen in figure

3.8a, as the speed at which the GPU is able to compute many terms is influenced by how many subterms are within them. This in turn is determined by how many times parameterised pivoting and local complementation occur, as well as how many phase nodes and phase pairs become separated from their diagram.

All the code relevant to this chapter is available at <https://github.com/mjsutcliffe99/ParamZX> (Sutcliffe, 2024a).

3.6 Further Applications

With regard to this thesis, the primary use of the parameterisation work outlined in section 3.2 is in application to classical simulation as described in section 3.4. Nevertheless, this work has additional applications outside of this scope, as this section briefly highlights.

3.6.1 Application to Circuit Measurements

Even without the GPU parallelism, the parameterised formalism of ZX-calculus presented in section 3.2 has a separate useful application, regarding quantum circuits with measurements. Specifically, basis measurements and corresponding probabilistic gates, used in fault tolerant circuits, may be expressed with Boolean parameters. It is then possible to simplify such circuits, largely parametrically, to, for instance, verify its behaviour across all possible measurement results.

An illustrative example follows. Figure 3.9, taken from (Kim, Baek, Hwang, & Bang, 2024), shows a quantum circuit implementing a fault-tolerant T-gate, with the explicit restriction of avoiding the inclusion of a probabilistic S gate ⁶.

where:

⁶The idea of applying this work to this example came from discussions with Harris Junseo Lee.

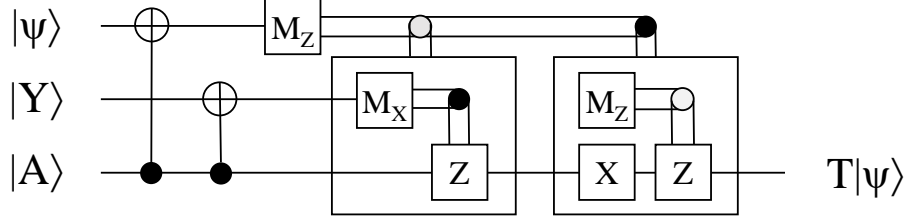


Figure 3.9: A quantum circuit implementing a fault tolerant T-gate without a probabilistic S gate correction (Kim et al., 2024).

$$\begin{aligned}
 |Y\rangle &:= SH|0\rangle = \text{red circle} - \text{yellow square} - \text{green circle } \frac{\pi}{2} \\
 |A\rangle &:= TH|0\rangle = \text{red circle} - \text{yellow square} - \text{green circle } \frac{\pi}{4}
 \end{aligned} \tag{3.69}$$

In this notation, M_Z and M_X denote measurements on the Z- and X- bases respectively. Doubled lines then indicate classical bits, as opposed to qubits, with a corresponding grey circle denoting the actions taken if this measurement returns 0 and a black circle if the measurement returns 1. For instance,

$$\begin{array}{c} \text{---} M_X \text{---} \\ \text{---} Z \text{---} \end{array} \equiv \begin{array}{c} \text{---} \text{grey circle } \text{b}\pi \\ \text{---} \text{grey circle } \text{b}\pi \end{array} \tag{3.70}$$

denotes an X-basis measurement with a probabilistic Z-gate that is present only if this measurement returns 1. As shown, this may be expressed in ZX-calculus form, with the inclusion of a parameter $\text{b} \in \mathbb{B}$. By extension,

$$\begin{array}{c} \text{---} M_Z \text{---} \\ \text{---} M_X \text{---} \\ \text{---} Z \text{---} \end{array} \equiv \begin{array}{c} \text{---} \text{red circle } \text{a}\pi \\ \text{---} \text{green circle } \text{b}\pi \\ \text{---} \text{green circle } (1 + \text{a})\text{b}\pi \end{array} \tag{3.71}$$

includes two such measurements, whose outcomes may be expressed with parameters, $\text{a}, \text{b} \in \mathbb{B}$. This sub-circuit also includes a probabilistic Z gate that is applied

only if $a = 0$ and $b = 1$. This is hence expressed as a π -phase Z-spider conditional on $(1 - a) \equiv (1 + a)$, enforcing $a = 0$, and b , enforcing $b = 1$.

By this reasoning, the circuit shown in figure 3.9 may be translated into the ZX-diagram shown in figure 3.10, where a conditional Hadamard gate:

$$\text{---} \boxed{a} \text{---} \equiv e^{ia\frac{\pi}{4}} \text{---} \left(\text{---} \frac{a\pi}{2} \text{---} \right) \left(\text{---} \frac{a\pi}{2} \text{---} \right) \text{---} = \begin{cases} \text{---} & \text{if } a = 0 \\ \text{---} \boxed{a} \text{---} & \text{if } a = 1 \end{cases} \quad (3.72)$$

is included to appropriately select the basis of the inner measurement, dependant upon the result of the outer (top-qubit) measurement:

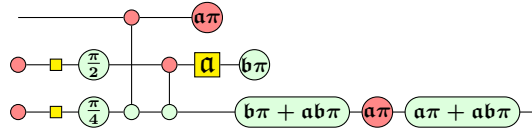


Figure 3.10: The circuit of figure 3.9 expressed as a parameterised ZX-diagram.

As this ZX-diagram contains, from the conditional Hadamard gate, fractional parameterised phases, it may not be fully simplified while maintaining generality without violating equation 3.21. However, by considering separately the two cases $a = 0$ and $a = 1$, it is possible to show, by reduction of just two parameterised ZX-diagrams, that this circuit implements a T-gate for all $a, b \in \{0, 1\}$.

The reduction of the $a = 0$ case follows:

(3.73)

For the sake of brevity, each step may involve multiple rewriting rules. The observations that $2b\pi = 0$ and $b = -b$, given $b \in \mathbb{B}^*$, are also used. The reduction of the $a = 1$ case follows likewise:

(3.74)

Likewise, a larger implementation, shown in figure 3.11, may be translated into the P-count 5 parameterised ZX-diagram shown in figure 3.12. Via the same approach as above, it may be verified, by reducing only two ZX-diagrams, that for

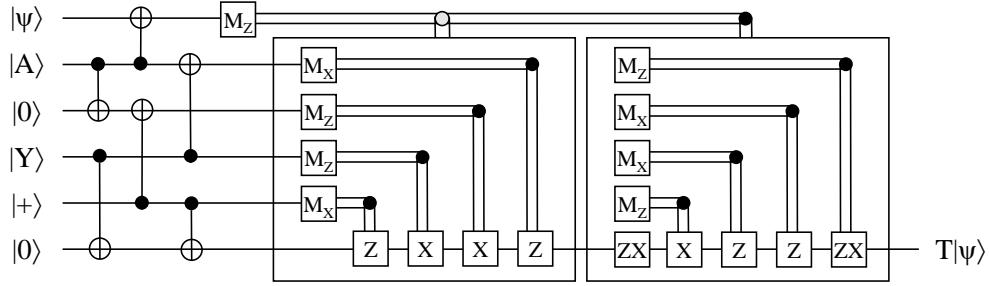


Figure 3.11: A quantum circuit implementing a fault tolerant T-gate without a probabilistic S gate correction (Fowler, 2012), as formulated in (Kim et al., 2024).

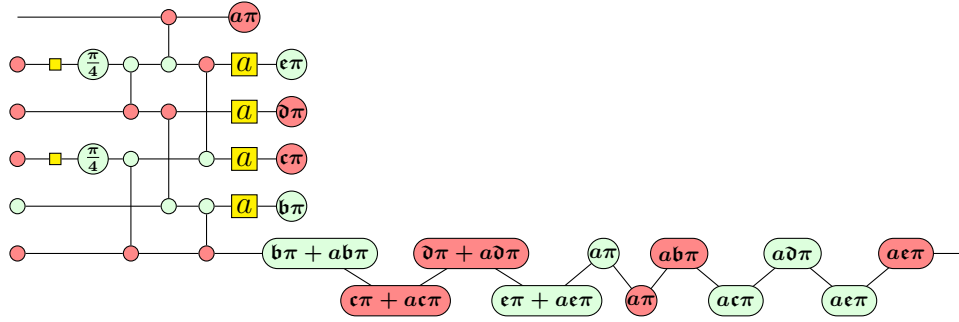


Figure 3.12: The circuit shown in figure 3.11 translated into a pair of parameterised ZX-diagrams, where $a \in \mathbb{B}$ and $b, c, d, e \in \mathbb{B}^*$.

all $2^5 = 32$ cases this circuit implements a T-gate. While the full derivation is excluded from this thesis, it is straightforward to show that in both the $a = 0$ and $a = 1$ cases, the diagram reduces and its parameters cancel, to ultimately result in a single T-gate.

Naturally, there is theoretically no limit to the number of measurements, N , that may be involved in a circuit, and with parameterised rewriting it is possible to verify the behaviour of such circuits by reducing $O(1)$ parameterised ZX-diagrams rather than all $O(2^N)$ distinct cases.

3.6.2 Parameterising Stabiliser Decompositions

One additional use of the parameterisation work outlined in this chapter is in parameterising stabiliser decompositions. Many such decompositions contain terms which are parametrically symmetric. This section highlights some examples.

Example 9. *The cutting decomposition (Codsì, 2022) of equation 2.68 may be parameterised as such:*

$$\begin{array}{c} \vdots \\ \textcircled{\alpha} \\ \vdots \end{array} = \frac{1}{\sqrt{2}^n} \sum_{a \in \mathbb{B}} e^{i\alpha a} \begin{array}{c} \textcircled{a\pi} \textcircled{a\pi} \\ \textcircled{a\pi} \textcircled{a\pi} \end{array} \quad (3.75)$$

Example 10. *The BSS decomposition (Bravyi et al., 2016) of equation 2.58 may be parameterised like so:*

$$\begin{aligned} e^{i\pi/4} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{4}} \\ | \end{array} = 2e^{i\pi/4} \begin{array}{c} \textcircled{-\frac{\pi}{2}} \end{array} \\ + \frac{1+\sqrt{2}}{4} \sum_{a \in \mathbb{B}} (-1)^a \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{a\pi} \\ | \end{array} \\ - 2\sqrt{2}i \sum_{b \in \mathbb{B}} \left(\frac{1}{\sqrt{2}}\right)^b \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\frac{\pi}{2}} \\ | \end{array} \\ + 8\sqrt{2}i \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \\ + 8\sqrt{2}i \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \begin{array}{c} | \\ \textcircled{\pi} \\ | \end{array} \end{aligned} \quad (3.76)$$

Example 11. *The work of (Koch et al., 2023) introduced a handful of new decompositions, shown in equation 2.66. Two of these may be parameterised to the*

following:

$$\begin{array}{c} \circ \\ | \\ \circ \\ | \\ \circ \end{array} = 3 \sum_{a \in \mathbb{B}} \left(-\frac{1}{3}\right)^a \begin{array}{c} a\pi \\ | \\ a\pi \\ | \\ a\pi \end{array} \quad (3.77)$$

$$+ \frac{3}{\sqrt{2}} \sum_{b \in \mathbb{B}} \left(-\frac{1}{2}\right)^b \begin{array}{c} b\pi \\ | \\ b\pi \\ | \\ b\pi \end{array}$$

$$\begin{array}{c} \pm\frac{\pi}{2} \\ | \\ \pm\frac{\pi}{2} \\ | \\ \pm\frac{\pi}{2} \end{array} = \frac{1}{2}(1 \pm 3i) \sum_{a \in \mathbb{B}} \left(\frac{-1 \pm 2i}{5}\right)^a \begin{array}{c} a\pi \\ | \\ a\pi \\ | \\ a\pi \end{array} \quad (3.78)$$

$$+ \frac{1}{2\sqrt{2}}(i - 3) \sum_{b \in \mathbb{B}} \left(\frac{-2 \pm i}{5}\right)^b \begin{array}{c} b\pi \\ | \\ b\pi \\ | \\ b\pi \end{array}$$

Depending on the specific use case, parameterising stabiliser decompositions in this way may be used to reduce the number of branches which need to be computed.

3.7 Conclusions

There are many applications of ZX-calculus where it is useful to compute (and often sum) many instances of the reduced scalar of a particular circuit, for various Boolean input/output bitstrings. This chapter highlights the redundancy inherent in such cases by demonstrating how the quantum circuit reduction strategies of ZX-calculus can be parameterised, with appropriate considerations, to allow circuits to be reduced while maintaining arbitrary Boolean inputs/outputs. In effect this means that instead of reducing a given circuit n times to compute its scalar for n different input/output bitstrings, one can reduce it just once (while maintaining generality), and efficiently *evaluate* the resulting parameterised expression n times using GPU hardware. Ultimately, it was shown that, applied to classical simulation, this led to an average speedup factor of 78.3 ± 10.2 .

While not implemented within the scope of this chapter, there are a number of

small techniques that could be employed to further optimise the method outlined here. For instance, after a given circuit has been reduced to a parameterised scalar, there may often be many simplifications that could be applied to minimise the number of subterms involved. For example, node-type subterms that share the same set of parameters but whose constant terms differ by π may be cancelled pairwise in place of a constant, such as:

$$\begin{aligned}
 (1 + e^{i\pi(\frac{2}{4}+(a\oplus b))})(1 + e^{i\pi(\frac{6}{4}+(a\oplus b))}) &= (1 + e^{i\pi\frac{2}{4}})(1 + e^{i\pi\frac{6}{4}}) \\
 &= (1 + i)(1 - i) \\
 &= 2
 \end{aligned} \tag{3.79}$$

(This is true because the result, in such cases, is the same regardless of whether $a \oplus b = 0$ or 1 .) A special case of this is in cancelling such pairs whose constants are 0 and π respectively, as this reduces overall to 0 , hence reducing the entire scalar term to 0 , negating any need to compute it further. This would increase the initial overhead time, and by extension the inflection point, but would reduce the number of subterms to compute and hence improve the terminal speedup factor.

Furthermore, while this chapter highlights a few key applications of this method, it remains very general and applicable in myriad areas. For instance, it is used again in later chapters, particularly in relation to ZX-diagram partitioning by vertex cutting in chapter 4. Conveniently, this method can typically be used in conjunction with, rather than instead of, other optimisations to classical simulation. For example, applied as it is in section 3.4, it is agnostic of the specific decompositions used, meaning it may still be used likewise as newer, more efficient decompositions are discovered. Moreover, it more effectively scales with the available hardware, as it is more feasible to increase the number of GPU cores manyfold than it is to increase the clockspeed of a CPU manyfold, particularly when utilising professional

computing clusters.

4 | Smarter ZX-Diagram Partitioning

This chapter presents a novel method by which Clifford+T ZX-diagrams may be reduced to scalar more efficiently, thereby enabling faster strong simulation of quantum circuits. In particular, this approach is based on partitioning ZX-diagrams into k smaller sub-diagrams which may each be independently reduced more efficiently. Thereafter, by appropriately cross-referencing these independent results, the overall scalar result can be attained, in (potentially) drastically fewer calculations than may be required otherwise.

After detailing this method and demonstrating example cases, the chapter also provides an analysis exploring the types of circuits upon which it is more effective, considering various qubit counts, depths, and degrees of interconnect- edness. Lastly, the chapter offers a means by which the method may be made generally more effective by employing the rewriting rules to re-express the initial ZX-diagrams in ways that better lend themselves to more efficient partitionings.

This chapter is based largely on the work presented in (Sutcliffe, 2024c).

4.1 Formalising the Existing Method

Lemma 18. *A parameterised scalar ZX-diagram:*

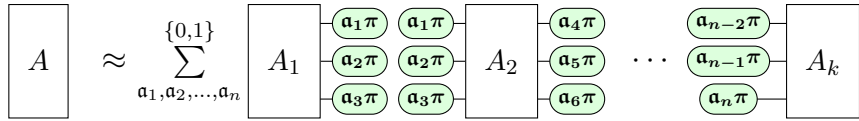
$$\begin{array}{c}
\boxed{G} \begin{array}{c} \text{---} \text{a}_1 \pi \\ \text{---} \text{a}_2 \pi \\ \vdots \\ \text{---} \text{a}_r \pi \end{array} \in {}^* \mathbb{C}
\end{array} \quad (4.1)$$

may be equivalently expressed as a function:

$$G(a_1, a_2, \dots, a_n) : \mathbb{B}^n \rightarrow \mathbb{C} \quad (4.2)$$

which reflects that the diagram may be fully reduced to a complex scalar, given some inputs: $a_i \in \mathbb{B} \forall i$.

Lemma 19. Consider a parameterised scalar ZX-diagram partitioned into k disjoint subgraphs:



$$A \approx \sum_{a_1, a_2, \dots, a_n}^{\{0,1\}} A_1 \begin{matrix} a_1 \pi \\ a_2 \pi \\ a_3 \pi \end{matrix} \begin{matrix} a_1 \pi \\ a_2 \pi \\ a_3 \pi \end{matrix} A_2 \begin{matrix} a_4 \pi \\ a_5 \pi \\ a_6 \pi \end{matrix} \cdots \begin{matrix} a_{n-2} \pi \\ a_{n-1} \pi \\ a_n \pi \end{matrix} A_k \quad (4.3)$$

As per lemma 18, this may be expressed in terms of functions:

$$A_i(\vec{\alpha}_i) : \mathbb{B}^{\dim(\vec{\alpha}_i)} \rightarrow \mathbb{C} \quad \forall i \in \{1, 2, \dots, k\} \quad (4.4)$$

like so:

$$\sum_{\vec{a} \in \mathbb{B}^n} \left[\prod_{i=1}^k A_i(\vec{\alpha}_i) \right] \quad (4.5)$$

where $\vec{a} = (a_1, a_2, \dots, a_n)$, with $a_j \in \mathbb{B} \forall j \in \{1, 2, \dots, n\}$, and:

$$\vec{\alpha}_i \subseteq \vec{a} \quad \forall i \in \{1, 2, \dots, k\} \quad (4.6)$$

Lemma 20. The runtime complexity for strong simulation of a Clifford+T quantum circuit via the existing method (Codsì, 2022) of ZX-diagram partitioning is:

$$O \left(2^c \sum_{i=1}^k 2^{\alpha t_i} \right) \quad (4.7)$$

where c is the number of parameters in the global ZX-diagram, k is the number of subgraphs into which it is partitioned, and t_i is the T-count of subgraph i . Meanwhile, α is the decomposition efficiency which, given the present state of the art (Qassim et al., 2021), is $\alpha \approx 0.396$.

Proof. The existing method in question is formalised mathematically in lemma 19, wherein expression 4.5 scales as $O(2^n k)$, given n (or c) total parameters among k partitioned subgraphs.

More precisely, each step involves computing a particular function $A_i(\vec{\alpha}_i)$, which corresponds to fully reducing an instance of the ZX-diagram described by A_i . From theorem 4, this itself is known to scale as $O(2^{\alpha t_i})$, given a T-count of t_i in subgraph A_i and decomposition efficiency α .

Lastly, as there are k such subgraphs to compute at each step, each with its own local T-count, the overall time complexity scales as stated in lemma 20. \square

Lemma 21. *Given an optimal partitioning for a particular k and c , the runtime complexity given in expression 4.7 simplifies to:*

$$O\left(2^{\frac{\alpha t}{k} + c} k\right) \quad (4.8)$$

where t is the **total** T-count of the global ZX-diagram and the remaining variables are as stated in lemma 20.

Proof. A k -partition via a given c cuts is optimal when the largest T-count subgraph is minimised. This is achieved when the local T-counts among the k subgraphs are as close to equal as possible: $t_i \approx t_j \forall i \in \{1, 2, \dots, k\}$. This in turn is attained when the overall T-count t is distributed evenly among them, such that

$t_i \approx \frac{t}{k} \forall i \in \{1, 2, \dots, k\}$. Hence:

$$O\left(2^c \sum_{i=1}^k 2^{\alpha t_i}\right) \rightarrow O\left(2^c \cdot k \cdot 2^{\alpha \frac{t}{k}}\right) \rightarrow O\left(2^{\frac{\alpha t}{k} + c} k\right) \quad (4.9)$$

□

Note, however, that this assumes a fixed k and c . In reality, these are **not** independent and finding an optimal partition in practice generally amounts to balancing these two metrics.

4.2 Redundancy Mitigation via Parameterisation

Section 2.4.3 covered the existing method — namely (Codsì, 2022) — relating to ZX-diagram partitioning for improved classical simulation. That section also highlighted some of the primary limitations of this method, such as k -partitions beyond $k \gtrsim 3$ being decreasingly likely to perform better than not partitioning at all.

In fact, as will be shown in this section, there is much redundancy among the calculations when this method is extrapolated for $k \geq 3$, and this redundancy can be mitigated at the cost of an increased memory overhead. Doing so resolves much of the issue causing $k \gtrsim 3$ partitions to be impractical.

In particular, recall lemma 5 which states that a ZX-diagram partitioned k -ways via c cuts ultimately produces $2^c \sum_{i=1}^k 2^{\alpha t_i}$ stabiliser terms, where t_i is the T-count of partitioned subgraph i and α is the stabiliser decomposition efficiency. This is, in fact, a naïvely motivated claim, providing an upper-bound result. In practice, provided $k \geq 3$, it is possible (and indeed likely) that each partitioned segment depends only on a reduced subset of the c cuts:

Lemma 22. Consider a ZX-diagram partitioned into $k \geq 3$ disjoint subgraphs via $c \equiv |\Gamma|$ cuts, denoted by parameters $\Gamma = \{\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_c\}$, where $\mathfrak{a}_i \in \mathbb{B}^* \forall i \in \{1, 2, \dots, c\}$.

Each such subgraph, i , depends upon a local subset of these global cut parameters:

$$\Gamma_i \subseteq \Gamma \quad \forall i \in \{1, 2, \dots, c\} \quad (4.10)$$

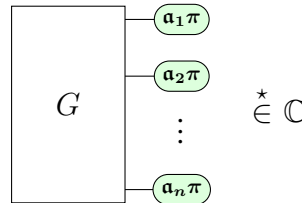
Hence, each subgraph, i , has:

$$2^{|\Gamma_i|} \leq 2^{|\Gamma|} \quad (4.11)$$

unique states¹, where $|\Gamma_i|$ denotes the cardinality of (i.e. number of elements in) Γ_i .

For each subgraph, one may pre-compute its unique states such that they can be recalled from memory without recalculation:

Definition 28. Given a parameterised scalar ZX-diagram $G(\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n)$:



$$\quad \quad \quad \in \mathbb{C}^* \quad (4.12)$$

where $G : \mathbb{B}^n \rightarrow \mathbb{C}$, its ‘**scalar profile**’ \vec{G} is a uniform tensor of rank n and length 2, housing all the scalars which may be returned by G , given its 2^n possible inputs:

$$\vec{G}[\vec{a}] = G(\vec{a}) \quad \forall \vec{a} \in \mathbb{B}^n \quad (4.13)$$

¹‘States’ in this context refer to static ZX-diagrams and not necessarily quantum states. Indeed, in general throughout this chapter, the term will refer to static *scalar* ZX-diagrams.

where:

$$\vec{a} = (a_1, a_2, \dots, a_n) \quad (4.14)$$

Note that, with an appropriate change of indexing, a scalar profile may likewise be treated as a linearised list of 2^n elements. In fact, this is how it is managed in memory and illustrated in examples later in the chapter.

Initialising the scalar profile of a ZX-diagram requires computing each unique scalar and hence has an associated runtime cost:

$$O(2^n) \quad (4.15)$$

given n Boolean free parameters.

This serves to replace each subgraph, $G_i([\Gamma_i]) : \mathbb{B}^{|\Gamma_i|} \rightarrow \mathbb{C}$, with a tensor, \vec{G}_i , storing its pre-computed unique scalars. While subtle, this change ensures that future calls of $G_i(\vec{a})$ can instead be exchanged for simply referencing the scalar result already recorded in memory: $\vec{G}_i[\vec{a}]$. The former requires fully reducing the subgraph via ZX-calculus decomposition which, as shown in previous chapters, is a very slow and computationally heavy process. The latter, meanwhile, is simply a retrieval of a complex scalar from memory and is hence trivial in its runtime cost.

Therefore, every unique state of each subgraph need only be computed once. Thus, the total number of calculations for strong simulation can potentially be significantly reduced:

Lemma 23. *Consider a ZX-diagram partitioned into k disjoint subgraphs via c total cuts, where each such subgraph i depends locally upon $c_i \leq c$ of these cuts and contains a T -count t_i .*

By pre-computing the unique states of each subgraph, the overall ZX-diagram may be strongly simulated by calculating:

$$S_{precomp} = \sum_{i=1}^k 2^{c_i} 2^{\alpha t_i} \quad (4.16)$$

stabiliser terms plus:

$$S_{crossref} = 2^c \quad (4.17)$$

calculations to appropriately cross-reference these unique sub-states.

Proof. From lemma 22 and theorem 4 it follows that each subgraph i has 2^{c_i} **unique** states which may be calculated via $2^{c_i} 2^{\alpha t_i}$ terms. Hence, given k such subgraphs, there exist $\sum_{i=1}^k 2^{c_i}$ unique sub-states, which may be pre-computed via:

$$\sum_{i=1}^k 2^{c_i} 2^{\alpha t_i} \leq 2^c \sum_{i=1}^k 2^{\alpha t_i} \quad (4.18)$$

terms.

Nevertheless, from lemma 5, the overall scalar result is given by summing over every combination of values of the global set of cut parameters $(a_1, a_2, \dots, a_c) \in \mathbb{B}^c$. This amounts to a further 2^c calculations to cross-reference the pre-computed sub-terms. \square

Hence, the overall number of calculations required has potentially reduced as, for ideal partitions (i.e. greater k with smaller $c_i \ll c \forall i$), the following inequality typically holds:

$$2^c + \sum_{i=1}^k 2^{c_i} 2^{\alpha t_i} \leq 2^c \sum_{i=1}^k 2^{\alpha t_i} \quad (4.19)$$

In essence, the computational cost of repeated calculation is mitigated by buffering the unique results. This comes at the cost of a memory overhead to store these unique sub-states, scaling as:

$$\sum_{i=1}^k 2^{c_i} \quad (4.20)$$

Ahead is an example to better illustrate how this optimisation works and to highlight its potential:

Example 12. Consider the following ZX-diagram which has been partitioned into 4 disjoint subgraphs, A, B, C, D , via 9 cuts:

$$\sum_{a,b,\dots,i}^{\{0,1\}} \begin{array}{c} \boxed{A} \begin{array}{cc} a\pi & a\pi \\ b\pi & b\pi \\ c\pi & c\pi \end{array} \boxed{B} \begin{array}{cc} d\pi & d\pi \\ e\pi & e\pi \\ f\pi & f\pi \end{array} \boxed{C} \begin{array}{cc} g\pi & g\pi \\ h\pi & h\pi \\ i\pi & i\pi \end{array} \boxed{D} \end{array} \quad (4.21)$$

In parameterised form, these 9 cuts manifest as 9 Boolean parameters, $a, b, \dots, i \in \mathbb{B}$, over which to sum, resulting in 2^9 summand terms as shown above.

This can equivalently be expressed algebraically as:

$$\sum_{a,b,\dots,i}^{\{0,1\}} A(a, b, c) \cdot B(a, b, c, d, e, f) \cdot C(d, e, f, g, h, i) \cdot D(g, h, i) \quad (4.22)$$

Notice that each subgraph depends only on some local subset of the 9 parameters. For instance, subgraph A depends only on three of these, namely parameters a, b, c .

In the first term, one has $a, \dots, h, i = 0, \dots, 0, 0$. This term is therefore given by:

$$A(0, 0, 0) \cdot B(0, 0, 0, 0, 0, 0) \cdot C(0, 0, 0, 0, 0, 0) \cdot D(0, 0, 0) \quad (4.23)$$

This means determining the scalar of subgraph $A(0, 0, 0)$ by first decomposing it to $2^{\alpha t_A}$ stabiliser terms, given a local T -count of t_A and decomposition efficiency α . The other three subgraphs are reduced likewise. Depending on these local T -counts, these steps alone can be computationally costly, requiring the computation of

$$2^{\alpha t_A} + 2^{\alpha t_B} + 2^{\alpha t_C} + 2^{\alpha t_D} \quad (4.24)$$

stabiliser terms just to compute this one summand term, given by expression 4.23.

The second such term, where $a, \dots, h, i = 0, \dots, 0, 1$, would next be calculated likewise as:

$$A(0, 0, 0) \cdot B(0, 0, 0, 0, 0, 0) \cdot C(0, 0, 0, 0, 0, 1) \cdot D(0, 0, 1) \quad (4.25)$$

At this point the redundancy may be obvious. In computing this next term, one must calculate via stabiliser decomposition each of: $A(0, 0, 0)$, $B(0, 0, 0, 0, 0, 0)$, $C(0, 0, 0, 0, 0, 1)$, and $D(0, 0, 1)$. However, from the previous term, the first two of these have already been calculated, as neither depends upon parameter i , which is the only parameter whose value has changed.

As such, one could avoid the costly steps of re-computing both $A(0, 0, 0)$ and $B(0, 0, 0, 0, 0, 0)$ every time they appear by keeping a record of the result the first time they are computed and simply recalling this result whenever needed.

More generally, for each subgraph, such as A , instead of performing full stabiliser decomposition 2^9 times (once for each combination of parameter values a, b, \dots, i), one can instead simply compute only the 2^3 **unique** results (in the case of A) by iterating over every combination of the **local** parameters (in this case: a, b, c). These results can then be stored and recalled as needed.

In this modest example case, this improvement reduces the number of (computationally costly) ZX-calculus reductions to compute from $4 \cdot 2^9 = 2048$ down to $2^3 + 2^6 + 2^6 + 2^3 = 144$. This is at the cost of a small memory overhead to store 144 scalars.

Altogether, this optimisation may be generalised like so:

Lemma 24. Consider a ZX-diagram, A , partitioned into k disjoint subgraphs, A_1, A_2, \dots, A_k :

$$A \approx \sum_{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n}^{\{0,1\}} A_1 \begin{matrix} \text{---} \text{a}_1 \pi \text{---} \text{a}_1 \pi \text{---} \\ \text{---} \text{a}_2 \pi \text{---} \text{a}_2 \pi \text{---} \\ \text{---} \text{a}_3 \pi \text{---} \text{a}_3 \pi \text{---} \end{matrix} A_2 \begin{matrix} \text{---} \text{a}_4 \pi \text{---} \\ \text{---} \text{a}_5 \pi \text{---} \\ \text{---} \text{a}_6 \pi \text{---} \end{matrix} \dots \begin{matrix} \text{---} \text{a}_{n-2} \pi \text{---} \\ \text{---} \text{a}_{n-1} \pi \text{---} \\ \text{---} \text{a}_n \pi \text{---} \end{matrix} A_k \quad (4.26)$$

where each subgraph contains a subset of the global Boolean parameters,

$\Gamma = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ where $\mathbf{a}_i \in \mathbb{B}^*$ $\forall i$:

$$\Gamma_i \subseteq \Gamma \quad \forall i \in \{1, 2, \dots, k\} \quad (4.27)$$

The full set of scalar profiles, $\{\vec{A}_2, \vec{A}_2, \dots, \vec{A}_k\}$, can be computed and recorded with a space and time complexity of:

$$O\left(\sum_{i=1}^k 2^{|\Gamma_i|}\right) \quad (4.28)$$

Then, the overall ZX-diagram A may be strongly simulated by computing:

$$\sum_{\vec{\alpha} \in \mathbb{B}^n} \left[\prod_{i=1}^k \vec{A}_i[\vec{\alpha}_i] \right] \quad (4.29)$$

which is a modification of equation 4.5 in lemma 19 to account for pre-computing subterms.

In summary, this optimisation is to recognise that, after partitioning, each subgraph is likely to depend upon only a reduced subset of the cut parameters. Consequently, when iterating over each unique *global* state, there is much repetition of unique *local* subgraph states. As calculating these subgraph states can be slow, significant speedup can be attained by pre-computing these unique subgraph states such that any redundancy here can be avoided, in exchange for an increased memory overhead.

4.3 GPU-Parallelised Cutting

One immediate improvement that can be made in this area is to apply the work proposed in chapter 3. Specifically, as formalised in lemma 5, when a Clifford+T ZX-diagram is partitioned k -ways via c cuts, one is left with 2^c Clifford+T terms which each then need be reduced (i.e. via stabiliser decomposition). As these partitions are induced via c instances of the vertex cutting decomposition, which (as outlined in chapter 3) is parametrically symmetric, these 2^c Clifford+T terms are themselves parametrically symmetric. Indeed, this is apparent by inspection of the general-case figure in lemma 5.

The natural implication of this observation is that the 2^c Clifford+T terms produced after partitioning may be computed in parallel batches on the GPU via the

means presented in chapter 3. Better still, this technique can be extended to account for the redundancy mitigation outlined in section 4.2. Where each subgraph $A(a_1, \dots, a_n)$ is pre-computed for every possible set of inputs, these 2^n cases can be computed in GPU-parallelised batches.

As noted in chapter 3, this improvement alone can yield a significant linear improvement to the runtime, particularly given enough cuts. In effect, the introduction of GPU parallelism here serves to speed up the rate, $R_{precomp}$, at which the unique subgraph scalars can be pre-computed.

4.4 Pairwise Partition Regrouping

Despite the optimisations outlined so far in this chapter, the number of calculations required for strong simulation via ZX-calculus still grows as $O(2^c)$ given c total cuts for partitioning. This is highlighted in lemma 23. In other words, there are still 2^c summand terms to compute after partitioning, but each such term can now be computed more rapidly. This limits runtime improvements to only linear speedups. However, as this section will detail, even the number of cross-referencing computations, and by extension the overall time complexity, can be exponentially reduced.

The idea is to iteratively regroup the pre-computed scalars of the partitioned subgraphs pairwise, rather than regrouping the collective at once. It is best explained by example:

Example 13. Recall the partitioned ZX-diagram from example 12:

$$\sum_{a,b,\dots,i}^{\{0,1\}} \begin{array}{c} \boxed{A} \begin{array}{cc} a\pi & a\pi \\ b\pi & b\pi \\ c\pi & c\pi \end{array} \boxed{B} \begin{array}{cc} d\pi & d\pi \\ e\pi & e\pi \\ f\pi & f\pi \end{array} \boxed{C} \begin{array}{cc} g\pi & g\pi \\ h\pi & h\pi \\ i\pi & i\pi \end{array} \boxed{D} \end{array} \quad (4.30)$$

which algebraically is:

$$\sum_{a,b,\dots,i}^{\{0,1\}} A(a,b,c) \cdot B(a,b,c,d,e,f) \cdot C(d,e,f,g,h,i) \cdot D(g,h,i) \quad (4.31)$$

Previously, each of the 2^9 summand terms here was computed iteratively. Consider instead initially just regrouping a single pair of subgraphs by summing over only the parameters common to both and ignoring the other parameters and subgraphs. For instance, one could regroup subgraphs A and B :

$$\sum_{a,b,c}^{\{0,1\}} A(a,b,c) \cdot B(a,b,c,d,e,f) = AB(d,e,f) \quad (4.32)$$

Recall that, after the initial pre-computation stage, each subgraph is stored no longer as a ZX-diagram but rather as a list of 2^{c_i} scalars, where c_i is the number of parameters (i.e. cuts) local to the subgraph i . As such, $A(a,b,c)$ is stored as a list of 2^3 scalars and $B(a,b,c,d,e,f)$ a list of 2^6 scalars. Given this, to ensure the result, $AB(d,e,f)$, is likewise recorded as a list of 2^3 scalars, rather than some many-termed parameterised expression, it is important to sum over all the parameters local to A and B , and not just those common to both. Hence, the number of calculations involved in this step is 2^p where p is the number of local parameters involved.

In this example, there are 6 parameters local to $A(a,b,c)$ and $B(a,b,c,d,e,f)$ and thus the two segments can be regrouped into $AB(d,e,f)$ in 2^6 calculations:

$$\sum_{d,e,\dots,i}^{\{0,1\}} \left[\begin{array}{c} \boxed{AB} \end{array} \begin{array}{cc} d\pi & d\pi \\ e\pi & e\pi \\ f\pi & f\pi \end{array} \begin{array}{c} \boxed{C} \end{array} \begin{array}{cc} g\pi & g\pi \\ h\pi & h\pi \\ i\pi & i\pi \end{array} \begin{array}{c} \boxed{D} \end{array} \right] \quad (4.33)$$

By the same means, one may then regroup $C(d,e,f,g,h,i)$ and $D(g,h,i)$ into

$CD(d, e, f)$ via 2^6 calculations:

$$\sum_{d,e,f}^{\{0,1\}} \left[\begin{array}{c} \boxed{AB} \quad \begin{array}{cc} \textcircled{d\pi} & \textcircled{d\pi} \\ \textcircled{e\pi} & \textcircled{e\pi} \\ \textcircled{f\pi} & \textcircled{f\pi} \end{array} \quad \boxed{CD} \end{array} \right] \quad (4.34)$$

This leaves one last iteration to regroup $AB(d, e, f)$ and $CD(d, e, f)$ into $ABCD$ via 2^3 calculations:

$$\boxed{ABCD} \quad (4.35)$$

With all segments now regrouped into one with no parameters remaining, the result is just a single scalar, equivalent to the original ZX-diagram. Thus, the initial circuit has been strongly classically simulated.

From these steps, one can observe that this example involved $2^6 + 2^6 + 2^3 = 136$ cross-reference calculations to fully regroup all the segments pairwise. This is as opposed to $2^9 = 512$ such calculations as would have been required to directly compute expression 4.31. Lastly, note that this is a modest example to illustrate the concept and that in practical cases this difference could be many orders of magnitude.

To generalise this, it is helpful to first define two new operators:

Definition 29. Given a set of sets $\Omega = \{A, B, \dots\}$, define the **collapsed set operator** \diamond like so:

$$(A \diamond B)_\Omega := (A \Delta B) \cup \left(A \cap B \cap \bigcup_{C \in \Omega \setminus \{A, B\}} C \right) \quad (4.36)$$

such that $(A \diamond B)_\Omega$ is the set containing:

- elements that are in A or B but not both, and

- elements that are in both A and B and at least one other set in the parent set Ω .

Definition 30. Given a set of sets $\Omega = \{A, B, \dots\}$, define the **exclusive internal operator** $\bar{\diamond}$ like so:

$$(A \bar{\diamond} B)_\Omega := (A \cap B) \setminus \bigcup_{C \in \Omega \setminus \{A, B\}} C \quad (4.37)$$

such that $(A \bar{\diamond} B)_\Omega$ is the set containing elements that are:

- in both A and B ,
- but **not** in any other set in the parent set Ω .

With these new operators, the procedure of example 13 may be expressed in general terms as follows:

Lemma 25. Consider a parameterised scalar ZX-diagram composed of two or more disjoint subgraphs, where $\Gamma_i \subseteq \Gamma$ is the set of parameters in subgraph G_i among the global set $\Gamma = \{\mathfrak{a}_1, \mathfrak{a}_2, \dots, \mathfrak{a}_n\}$ where $\mathfrak{a}_i \stackrel{*}{\in} \mathbb{B} \ \forall i$.

Define the scalar profile \vec{G}_i as a tensor of rank $|\Gamma_i|$ associated with subgraph G_i such that:

$$\vec{G}_i[\vec{x}] = G_i(\vec{x}) \quad \forall \vec{x} \in \mathbb{B}^{|\Gamma_i|} \quad (4.38)$$

Any pair of the subgraphs, G_A and G_B , may be effectively regrouped by populating a new scalar profile \vec{G}_{AB} for its combined subgraph G_{AB} like so:

$$\vec{G}_{AB}[\vec{z}] = \sum_{\vec{w} \in \mathbb{B}^{\dim(\vec{w})}} \vec{G}_A[\vec{x}] \times \vec{G}_B[\vec{y}] \quad \forall \vec{z} \in \mathbb{B}^{\dim(\vec{z})} \quad (4.39)$$

where:

$$\begin{aligned}\vec{x} &= [\Gamma_A] \\ \vec{y} &= [\Gamma_B]\end{aligned}\tag{4.40}$$

and:

$$\vec{z} = [\Gamma_A \Delta \Gamma_B]\tag{4.41}$$

$$\vec{w} = [\Gamma_A \cap \Gamma_B]\tag{4.42}$$

if using edge cuts, though — more generally and more powerfully — if using vertex cuts:

$$\vec{z} = [(\Gamma_A \diamond \Gamma_B)_\Omega]\tag{4.43}$$

$$\vec{w} = [(\Gamma_A \bar{\diamond} \Gamma_B)_\Omega]\tag{4.44}$$

and:

$$\Omega = \{\Gamma_A, \Gamma_B, \Gamma_C, \dots\}\tag{4.45}$$

Example 14. Consider a ZX-diagram, containing parameters $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d} \stackrel{\star}{\in} \mathbb{B}$, composed of a chain of three partitioned subgraphs, A, B, C , of which one aims to regroup A and B :

$$\sum_{b \in \mathbb{B}} \textcircled{a\pi} \text{---} \boxed{A} \text{---} \textcircled{b\pi} \text{---} \textcircled{b\pi} \text{---} \boxed{B} \text{---} \textcircled{c\pi} \text{---} \textcircled{c\pi} \text{---} \boxed{C} \text{---} \textcircled{d\pi} = \textcircled{a\pi} \text{---} \boxed{AB} \text{---} \textcircled{c\pi} \text{---} \textcircled{c\pi} \text{---} \boxed{C} \text{---} \textcircled{d\pi} \quad (4.46)$$

Here, each subgraph contains just two parameters:

$$\Gamma_A = \{\mathfrak{a}, \mathfrak{b}\}, \quad \Gamma_B = \{\mathfrak{b}, \mathfrak{c}\}, \quad \Gamma_C = \{\mathfrak{c}, \mathfrak{d}\} \quad (4.47)$$

and thus the three corresponding scalar profiles, $\vec{A}_{a,b}, \vec{B}_{b,c}, \vec{C}_{c,d}$, may each be recorded as a 2×2 matrix of complex scalars.

Next, note that the prospective regrouped AB subgraph will contain parameters:

$$\Gamma_{AB} = (\Gamma_A \diamond \Gamma_B)_\Omega = (\{\mathfrak{a}, \mathfrak{b}\} \diamond \{\mathfrak{b}, \mathfrak{c}\})_\Omega = \{\mathfrak{a}, \mathfrak{c}\} \quad (4.48)$$

where $\Omega = \{\Gamma_A, \Gamma_B, \Gamma_C\}$.

Then, the regrouped \overrightarrow{AB} may be initialised as a rank $|\Gamma_{AB}|$ tensor and populated accordingly:

$$\overrightarrow{AB}[a, c] = \sum_{b \in \mathbb{B}} \vec{A}[a, b] \times \vec{B}[b, c] \quad \forall (a, c) \in \mathbb{B}^2 \quad (4.49)$$

This will involve iterating over all:

$$\Gamma_A \cup \Gamma_B = \{\mathfrak{a}, \mathfrak{b}\} \cup \{\mathfrak{b}, \mathfrak{c}\} = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\} \quad (4.50)$$

for a total of $2^{|\{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\}|} = 2^3$ cross-reference calculations.

a	b	c	A_{ab}	B_{bc}	$(AB)_{abc}$	$\sum_b^{\{0,1\}} (AB)_{abc}$	$\equiv (AB)_{ac}$
0	0	0	A_{00}	B_{00}	$A_{00}B_{00}$	$(A_{00}B_{00} + A_{01}B_{10})$	$\equiv (AB)_{00}$
0	0	1	A_{00}	B_{01}	$A_{00}B_{01}$	$(A_{00}B_{01} + A_{01}B_{11})$	$\equiv (AB)_{01}$
0	1	0	A_{01}	B_{10}	$A_{01}B_{10}$		
0	1	1	A_{01}	B_{11}	$A_{01}B_{11}$		
1	0	0	A_{10}	B_{00}	$A_{10}B_{00}$	$(A_{10}B_{00} + A_{11}B_{10})$	$\equiv (AB)_{10}$
1	0	1	A_{10}	B_{01}	$A_{10}B_{01}$	$(A_{10}B_{01} + A_{11}B_{11})$	$\equiv (AB)_{11}$
1	1	0	A_{11}	B_{10}	$A_{11}B_{10}$		
1	1	1	A_{11}	B_{11}	$A_{11}B_{11}$		

Table 4.1: The (linearised) scalar profiles, A_{ab} and B_{bc} , may be regrouped into $(AB)_{ac}$ by iterating over the parameters involved and cross-referencing the scalars accordingly.

Essentially, for each $(a, b, c) \in \mathbb{B}^3$, one may retrieve the scalars $A(a, b)$ and $B(b, c)$ and multiply them to deduce the scalar $AB(a, b, c)$. In turn, this newly calculated scalar may be added to the relevant $AB(a, c)$. In other words, after computing all products $AB(a, b, c) \forall (a, b, c)$, the resulting list may be reduced to $AB(a, c)$ by summing $\sum_b^{\{0,1\}} AB(a, b, c) = AB(a, c)$ for each (a, c) . This is illustrated in table 4.1, and high-level pseudocode that implements this procedure is shown in algorithm 5.

The simple examples discussed so far have been of neatly partitioned chains of segments. Realistically, however, efficiently partitioning a graph k -ways is likely to result in more chaotic and intertwined segment connections. In this context, a connection between (or among) segments represents a cut that has been made which gave rise to a parameter common to these segments. Moreover, as it is vertices (i.e. spiders) rather than edges that are being cut, it is possible for a cut to affect (and hence introduce a new parameter to) more than two segments. Example 15 demonstrates a slightly more realistic case.

Example 15. *The following equation shows an example of a partitioned ZX-diagram undergoing one instance of pairwise regrouping:*

$$\sum_{a,b,\dots,l}^{\{0,1\}} = \sum_{d,e,\dots,l}^{\{0,1\}} \quad (4.51)$$

In this example, the initial subgraphs are:

- $A(a, b, c, d, e, f)$
- $B(a, b, c, d, g, h, i, j, k)$
- $C(d, e, f, g, h, k, l)$
- $D(i, j, k, l)$

such that, of the overall set of parameters $\Gamma = \{a, b, \dots, l\}$, each subgraph depends upon some subset of Γ :

- $\Gamma_A = \{a, b, c, d, e, f\}$
- $\Gamma_B = \{a, b, c, d, g, h, i, j, k\}$
- $\Gamma_C = \{d, e, f, g, h, k, l\}$
- $\Gamma_D = \{i, j, k, l\}$

The process shown in equation 4.51 is the regrouping of subgraphs $A(a, b, c, d, e, f)$

and $B(a, b, c, d, g, h, i, j, k)$ into a collective $AB(d, e, f, g, h, i, j, k)$, where:

$$\Gamma_{AB} = (\Gamma_A \diamond \Gamma_B)_\Omega = \{d, e, f, g, h, i, j, k\} \quad (4.52)$$

given $\Omega = \{\Gamma_A, \Gamma_B, \Gamma_C, \Gamma_D\}$.

Notice that the parameters common to A and B but which appear nowhere else:

$$(\Gamma_A \bar{\diamond} \Gamma_B)_\Omega = \{a, b, c\} \quad (4.53)$$

are completely removed when A and B are regrouped.

Consequently, the overall set of parameters in the system is reduced according to:

$$\Gamma \leftarrow \Gamma \setminus (\Gamma_A \bar{\diamond} \Gamma_B)_\Omega \quad (4.54)$$

such that, after regrouping, $\Gamma = \{d, e, f, g, h, i, j, k, l\}$.

Algorithm 5 details the procedure for performing an iteration of such pairwise regrouping. This is high-level pseudocode, making use of high-level data structures (such as *sets*) and functions (such as those pertaining to set theory and the handling of bitstrings). While this aims to convey how the procedure works in broad, and hopefully easy to follow, terms, it does mean this implementation is sub-optimal for the time sensitive nature of its use case. Consequently, listing 4.1 presents a far more efficient, low-level implementation of this procedure, making use of binary encoding, bitwise calculations, and GPU parallelism.

Algorithm 5 A high-level implementation of pairwise regrouping

```

1: function REGROUPPAIR( $\vec{A}, \vec{B}$ )
2:    $commonParams \leftarrow \vec{A}.localParams \cup \vec{B}.localParams$ 
3:    $exclusiveParams \leftarrow \vec{A}.localParams \diamond \vec{B}.localParams$ 
4:    $n \leftarrow \text{length}(commonParams)$ 
5:    $m \leftarrow \text{length}(exclusiveParams)$ 
6:
7:    $\vec{AB} \leftarrow [0, 0]^m$ 
8:    $\vec{AB}.localParams \leftarrow exclusiveParams$ 
9:   for all  $(a, b, c, \dots) \in \mathbb{B}^n$  do
10:     $\vec{x} \subseteq \{(a, b, c, \dots) | (a, b, c, \dots) \in \vec{A}.localParams\}$ 
11:     $\vec{y} \subseteq \{(a, b, c, \dots) | (a, b, c, \dots) \in \vec{B}.localParams\}$ 
12:     $\vec{z} \subseteq \{(a, b, c, \dots) | (a, b, c, \dots) \in exclusiveParams\}$ 
13:     $\mathfrak{a} \leftarrow a, \mathfrak{b} \leftarrow b, \mathfrak{c} \leftarrow c, \dots$ 
14:
15:     $A_{ab} \leftarrow \vec{A}.scalars[\vec{x}]$ 
16:     $B_{bc} \leftarrow \vec{B}.scalars[\vec{y}]$ 
17:     $AB_{abc} \leftarrow A_{ab} \times B_{bc}$ 
18:
19:     $\vec{AB}[\vec{z}] \leftarrow \vec{AB}[\vec{z}] + AB_{abc}$ 
20:   end for
21:    $\vec{A} \leftarrow \vec{AB}$ 
22:    $\vec{B} \leftarrow \text{null}$ 
23: end function

```

Listing 4.1: An efficient, low-level and GPU-parallelised CUDA kernel for pairwise regrouping.

```

__global__ void regroup_pair_gpu(int paramsA, int paramsB, int
    paramsC, float * A_re, float * A_im, float * B_re, float *
    B_im, float * AB_re, float * AB_im, const int N_params, const
    int size)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    // LOCALLY INDEX...

```

```

int ab = 0;
int bc = 0;
int ac = 0;
int abc = index;
int x = 0; // current length of ab
int y = 0; // current length of bc
int z = 0; // current length of ac

for (int i=0; i<N_params; ++i)
{
    if (paramsA & 1) ab = ((abc & 1) << x++) | ab;
    if (paramsB & 1) bc = ((abc & 1) << y++) | bc;
    if (paramsC & 1) ac = ((abc & 1) << z++) | ac;
    abc >>= 1;
    paramsA >>= 1;
    paramsB >>= 1;
    paramsC >>= 1;
}

// MULTIPLY SCALARS (A_ab * B_bc -> AB_abc) ...
// (A+ai)(B+bi) = (AB-ab) + (Ab+aB)i
float A = A_re[ab];
float a = A_im[ab];
float B = B_re[bc];
float b = B_im[bc];

//AB_re[index] = (A*B) - (a*b):
atomicAdd(&AB_re[ac], (A*B) - (a*b));
//AB_im[index] = (A*b) + (a*B):
atomicAdd(&AB_im[ac], (A*b) + (a*B));
__syncthreads();
}

```

This CUDA code shows the function `regroup_pair_gpu`. Be aware that this is a CUDA kernel, rather than a conventional function, meaning it is executed many times in parallel upon the GPU threads. While this kernel code is C-like, it may be called from within Python, passing as arguments the local parameter sets of segments A and B respectively, together with their exclusively uncommon parameters (i.e. those of the future grouped AB segment), and also the total number of parameters involved among A and B .

Rather than using a high-level data structure like a *set* or even a *list*, this instead records sets of parameters as individual integers. The example of table 4.1 involved a segment A containing parameters $\{a, b\}$ and a segment B containing parameters $\{b, c\}$. So, for a collective set of parameters $\{a, b, c\}$, this here would be expressed as `paramsA = 110` and `paramsB = 011`. (As integers this would be interpreted as `paramsA = 6` and `paramsB = 3`, but for the purposes here it makes more sense to interpret these as their binary bitstrings.) This example case would also include `paramsC = 101` being the exclusive uncommon parameter set (i.e. $\{a, c\}$, which will be the parameters in the upcoming regrouped segment, AB). Converting the sets into this form can be done quite trivially in Python in advance of calling the kernel, and then this data can be passed among its arguments, along with the total number of parameters involved (in this case, A and B collectively contain 3 parameters, $\{a, b, c\}$, so `N_params = 3`). Meanwhile, the argument `size` is just the number of parallel threads to compute (i.e. the number of rows of table 4.1), 2^{N_params} . (Note that the remaining arguments of the kernel, such as `A_re`, refer to the memory wherein the real and imaginary parts of the list of scalars of segments A and B are recorded, while those of `AB_re` and `AB_im` are initially just empty blocks of data, acting as empty arrays to which the kernel will be writing.)

The only difference among each parallel thread executing this kernel is its unique

identifier number, `index`, ranging from 0 to `size-1`. This `index` essentially gives the *values* of the parameters for this thread. For instance, in the example case, `index=5` — which in binary would be 101 — would denote the case whereby $a = 1$, $b = 0$, $c = 1$. (This is the $(\text{index}+1)^{\text{th}}$ row of table 4.1.) The logic of the `for` loop in this kernel then serves to take this `index`, representing the full set of parameters, $\{a, b, c\}$, and determine the respective *local* sets of parameters of segments A and B — in this case $\{a, b\}$ and $\{b, c\}$. (So, for `index=5`, this would take `abc=101` and deduce `ab=10` and `bc=01`.) This works via clever usage of bitshifting and bitwise operations, and — for the keen reader — the best way to understand the logic is to work through the table 4.1 example step-by-step.

For each (parallel) iteration of the kernel, one scalar, AB_{ac} , among the new re-grouped segment will be calculated and saved to memory. The only remaining point to note here is the use of the `atomicAdd(x, y)` CUDA function. This adds the value `y` to the memory address `x`, but does so in a parallel-friendly way which avoids race conditions.

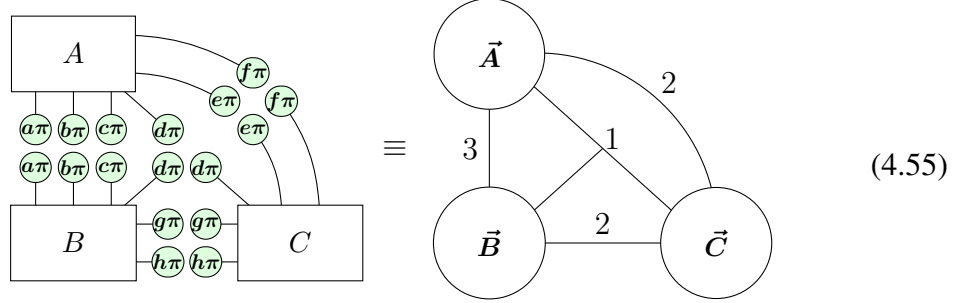
Regarding its use in the ZX-Partitioner, if the projected runtime is below a certain threshold then the high-level implementation of algorithm 5 is used (coded in Python), as on such low scales the overhead in initialising the data to the kernel makes the GPU approach actually slower than the higher-level implementation. However, for sufficiently (non-trivially) sized cases, the efficient CUDA implementation is used, giving a drastic performance speedup (for this particular part of the computations).

4.5 The ZX-Partitioner

The concept so far may be summarised as follows:

1. Start with an initial static scalar ZX-diagram, G , and apply Clifford simplification (Kissinger & van de Wetering, 2020b) to transform it to reduced gadget form.
2. Use an existing hypergraph partitioning algorithm (Schlag, 2020) and the vertex cutting decomposition (equation 2.68) to partition G into k disjoint parameterised ZX-diagrams, G_1, \dots, G_k , of approximately equal T-counts.
3. For each parameterised ZX-diagram $G_i(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \dots)$ pre-compute its scalar profile, \vec{G}_i . That is, for each i , compute and record every possible scalar $\vec{G}_i[a, b, c, \dots] = G_i(a, b, c, \dots) \forall a, b, c, \dots \in \mathbb{B}$.
4. Regroup these k scalar profiles pairwise, $\vec{G}_i, \vec{G}_j \rightarrow \overrightarrow{G_i G_j}$, until all have been reduced to a single rank 0 scalar profile, $\overrightarrow{G_1 G_2 \dots G_k}$. This is equal a single scalar, which is the final result, equivalent to reducing the original ZX-diagram G .

Essentially, the initial ZX-diagram is divided into parameterised subgraphs of manageable T-counts, which are reduced, for all parameter values, via stabiliser decomposition. These resulting *scalar profiles*, are simply tensors storing all possible scalars of each subgraph. These may be reduced via a tensor contraction approach to deduce the final overall scalar result. It is helpful, after the initial ZX-diagram has been partitioned and the scalar profiles of each subgraph deduced, to thereafter treat the system as a tensor network, which may be expressed as a hypergraph, as in the following example:



This should not be confused with the hypergraph representation of a ZX-diagram from section 2.4.3. Rather, each segment, \vec{A} , \vec{B} , \vec{C} , here denotes the scalar profile of the corresponding subgraph, A, B, C . For ease of visualisation, where there are multiple hyperedges connecting a set of nodes, these may be illustrated with a single hyperedge, labelled with the number of such hyperedges it represents.

Given the goal is to eventually regroup all segments together, to do this efficiently (that is, minimising the number of calculations involved), at each step, the pair selected for regrouping should collectively have the fewest number of local parameters (i.e. hyperedges). For a hypergraph, H , containing k nodes (i.e. segments), this number will be given by the function $\text{minpair}(H)$, which can be computed in $O(k^2)$ time (and as k is always relatively low, this runtime is generally negligible).

Given the methods presented in this chapter, the number of computations required to fully reduce a t T-gate ZX-diagram to its scalar has been brought down from $2^{\alpha t}$ to a potentially much more modest $S_{precomp} + S_{crossref}$ (plus some negligible overhead from the partitioning function itself), where:

$$\begin{aligned}
 S_{precomp} &= \sum_{i=1}^k 2^{\alpha t_i + c_i} \\
 S_{crossref} &= \sum_{i=0}^{k-2} 2^{\text{minpair}(H_i)}
 \end{aligned}
 \tag{4.56}$$

The first equation here describes the computational cost of precomputing the unique scalars of each partitioned segment. i iterates through each segment, such that t_i and c_i are its local T-count and local parameter (i.e. hyperedge) count respectively. Meanwhile, the second equation describes the computational cost of cross-referencing these precomputed scalars (as in table 4.1). In other words, it is the cost of regrouping the partitioned segments. Here, i denotes the regrouping *step* - that is, $i = 0$ refers to the initial hypergraph state (with k segments) and each successive step ($i \rightarrow i + 1$) is defined by when the next cheapest pair of segments is regrouped (reducing the number of segments by one: $k \rightarrow k - 1$). H_i , therefore, is the state of the hypergraph after i instances of pairwise regrouping.

The culmination of this work is a new Python package called the *ZX-Partitioner* (Sutcliffe, 2024d), which may be found on Github at: <https://github.com/mjsutcliffe99/zxpartitioner>. At its most abstracted, this package offers a function into which the user may provide a ZX-diagram (a *graph* from the *PyZX* package (Kissinger & van de Wetering, 2020a, 2018)) and its scalar equivalent will be calculated using the methods outlined in this chapter. For convenience and to help the reader/user better understand the methodology, this routine can also be run one phase at a time and with visualisations of the partitioned ZX-diagrams and, more helpfully, their segment connectivity hypergraph at each step. It may also be easily configured for different stabiliser state decomposition strategies and hardware capabilities.

Of this main function, the initial step performed is to determine the most efficient number of parts, k , into which the ZX-diagram should be partitioned. Partitioning into a greater number of parts means that each will be of a lower T-count and hence the number of *precomputing* calculations, $S_{precomp}$, will be drastically reduced. (For a given graph, the typical number of *local* cuts, c_i , on each part, i , tends to not vary too drastically regardless of k , and at any rate it is likely to be much

smaller than αt_i , so the local T-count tends to be the significant contributor to $S_{precomp}$. This is especially true when considering the projected *runtimes* rather than the number of computations, as each computation in the 2^{c_i} component can be computed much more rapidly than those in the $2^{\alpha t_i}$ component, as highlighted in section 4.3.) However, taking a larger k comes at the cost of increasing the number of *total* cuts, C . While ordinarily this would render larger k values infeasible, as shown in section 4.4 this need not be the case. Nevertheless, taking a larger k indeed increases the number of cross-reference calculations, $S_{crossref}$, albeit not so drastically. This is because when a pair of segments, A and B , is regrouped, the resulting segment will have a number of local cuts, c_{AB} , equal to the number of cuts in the symmetric difference of c_A and c_B . This in turn means that as more segments are regrouped pairwise, there is a higher likelihood of segments having larger numbers of local cuts, which results in a larger $\text{minpair}(H)$.

Put concisely, partitioning into a larger k results in decreasing $S_{precomp}$ but increasing $S_{crossref}$. As the *overall* number of computations is given by the sum of these two, then the most optimal k is that which produces the crossover point where these two terms are as close to equal as possible, such that neither dominates and renders the other negligible. (Note that the k -partitioning function itself generally runs in negligible time.) Fortunately, for any k , $S_{precomp}$ and $S_{crossref}$ can be determined in advance in negligible time. Consequently, the optimal choice of k can likewise be determined in advance in negligible time. (In fact, balancing the projected *runtimes*, $T_{precomp}$ and $T_{crossref}$, as discussed in the following section, yields even better results.)

With the optimal k determined, the next step is to k -partition the ZX-diagram, which one can visualise as a hypergraph of partitioned segments with connected edges representing common cut parameters, as in figure 4.1.

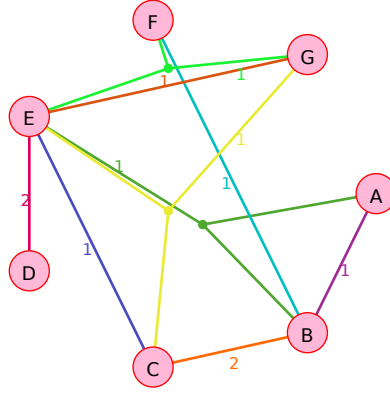


Figure 4.1: An example of a segment connectivity hypergraph, generated via the *ZX-Partitioner*.

At this point, the segments may be precomputed — in each case, i , turning a parameterised ZX-diagram (of c_i parameters, arising from c_i local cuts) into a list of 2^{c_i} scalars. c_i here also denotes the number of edges connected to the particular segment, i , in the hypergraph (figure 4.1).

Next, the program will find the pair of connected segments, A and B , with the fewest collective number of local edges, $c_A + c_B$. This will be the cheapest connected pair to regroup and so regrouped it is, into segment AB (as detailed in section 4.4). Having fused these two segments together, the hypergraph will now contain one fewer segment in total. This step may then be repeated, regrouping whichever connected pair of segments is *now* the cheapest. This process continues until the final two segments are regrouped into one. Figure 4.2 shows this in action.

This final segment will have no edges (i.e. local cuts) and hence will record a single (as $2^0 = 1$) scalar. This scalar is the final result, which is equivalent to the original scalar ZX-diagram.

Lastly, note that the example of figure 4.1 is a very simple case for illustrative purposes. More realistic examples are shown in figure 4.3.

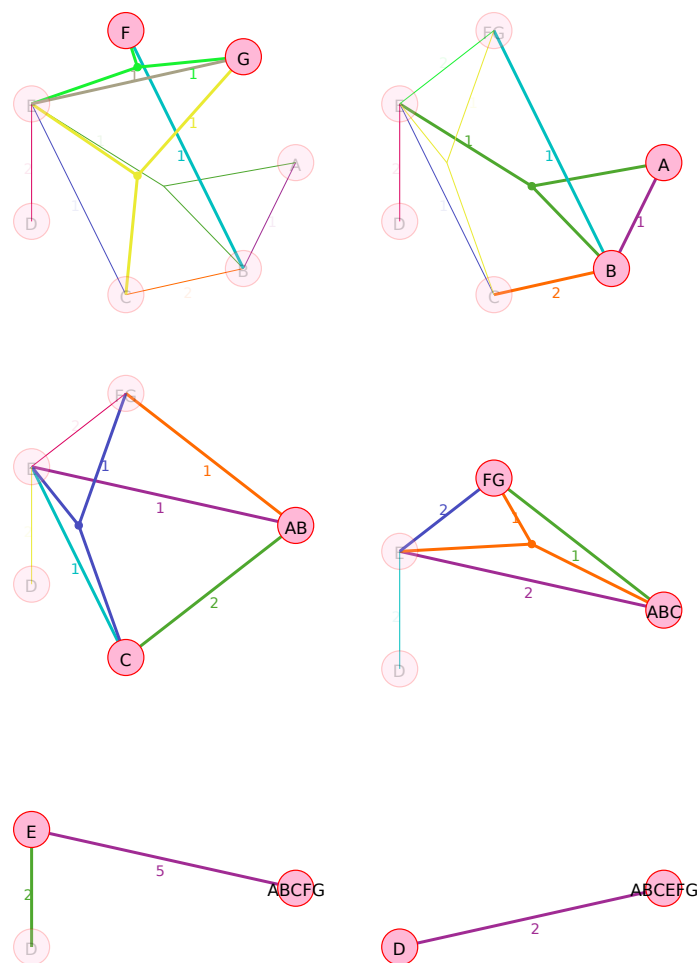


Figure 4.2: The precomputed segments of a partitioned ZX-diagram may be regrouped pairwise (selecting the cheapest pair to regroup at each step) until one segment remains. The steps in this figure are shown chronologically in row-major order. In each case, the local edges among the cheapest pair are highlighted, with the sum of their weights, w , giving the computational cost of regrouping, 2^w . Regrouping the final remaining pair will provide the overall scalar result. (Note that the edge colours are random and exist for visual clarity but bear no meaning.)

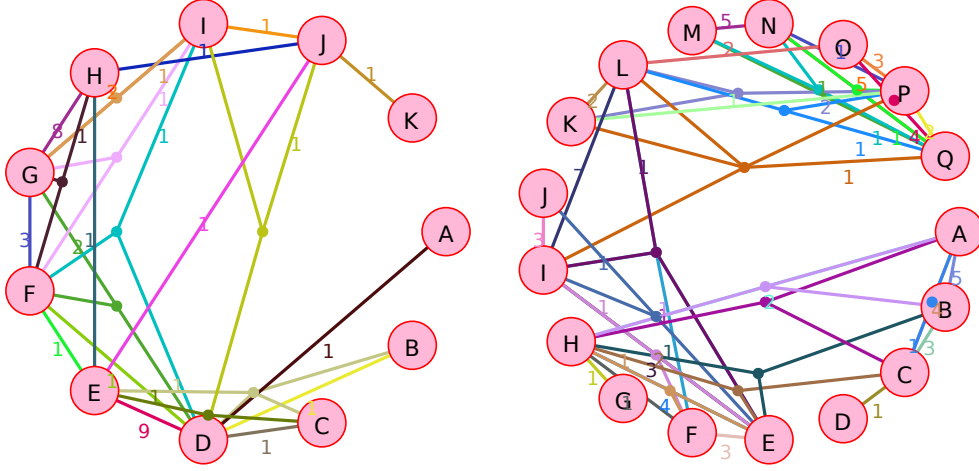


Figure 4.3: Two heftier examples of segment connectivity hypergraphs, generated from partitioning random ZX-diagrams via the ZX-Partitioner.

4.6 Estimating Runtime

4.6.1 Direct Decomposition

Section 2.3.4 described the existing non-partitioning approach to strong simulation via ZX-calculus, as introduced in (Kissinger & van de Wetering, 2022). As outlined therein, the runtime of this method is proportional to the number of stabiliser terms produced, which itself grows exponentially with the initial T-count. This information can be used to quantify an estimate of the runtime of such a method:

Lemma 26. *The runtime of strong simulation via stabiliser decomposition of ZX-diagrams **without** the use of partitioning (such as the methods of (Kissinger & van de Wetering, 2022; Kissinger et al., 2022)) can be approximated as:*

$$T_{decomp} \approx \frac{S_{decomp}}{R_{decomp}} \quad (4.57)$$

where $S_{decomp} = 2^{\alpha t}$, given decomposition efficiency α and initial T-count t , is the the number of stabiliser terms to compute. R_{decomp} is then the average rate at which such terms can be computed, measured in terms per second.

The average computation rate, R_{decomp} , can be determined experimentally by measurement across many instances of ZX-diagram reduction. For the purposes of this thesis, this rate was measured for over a hundred randomly generated circuits of various practical T-counts, qubit counts, depths, and partitionabilities, giving an average of:

$$R_{decomp} = 1,730 \pm 650 \text{ calcs/s} \quad (4.58)$$

These measurements, quoted to 3 significant figures with error margins given by standard deviation, were recorded on a commercial laptop with an *11th Gen Intel Core i5-11400H @ 2.70GHz CPU, NVIDIA GeForce GTX 1650 GPU, and 8GB SODIMM RAM.*

Unsurprisingly, these measurements exhibited variance across the circuit metrics (depth, T-count, etc.), contributing to this notable 38% error margin. However, this is not a problem for the concerns of this thesis, which is focused on approximate scale and order of magnitude, for comparison of methods, rather than precise runtime figures (which, regardless, would depend upon hardware). As such, the above figure for R_{decomp} can be taken as approximately accurate for any non-trivial circuit given the hardware used.

4.6.2 Smart Partitioning

On the other hand, the method outlined in section 4.2 separates the computations into two parts: pre-computing the unique subgraph scalars and cross-referencing

these results to attain the overall scalar result. The overall runtime in this case then conforms to the following:

Lemma 27. *Given the number of calculations, being $S_{precomp} + S_{crossref}$ as per lemma 23, the overall runtime for strong simulation via such ZX-diagram partitioning can be approximated as:*

$$T_{total} \approx T_{overhead} + \frac{S_{precomp}}{R_{precomp}} + \frac{S_{crossref}}{R_{crossref}} \quad (4.59)$$

where $R_{precomp}$ and $R_{crossref}$ are the calculation **rates** (number of calculations computable per second) for the pre-computation and cross-referencing stages respectively. $T_{overhead}$, meanwhile, is the initialisation time to partition the ZX-diagram.

As before, the computation rates for each part can be measured experimentally across a wide range of random examples. Given the same restrictions and hardware as above, the average measurements were:

$$R_{precomp}^{cpu} = 1,730 \pm 650 \text{ calcs/s} \quad (4.60)$$

$$R_{crossref}^{cpu} = 148,000 \pm 34,000 \text{ calcs/s} \quad (4.61)$$

The overhead time, $T_{overhead}$, was measured similarly and found to vary significantly with the degree to which the given ZX-diagram was partitioned. Nevertheless, even in the most extreme cases this seldom exceeded a few seconds (and generally was well below one second). As such, for all bar the trivially small and rapid cases, $T_{overhead}$ represented a negligible contribution to the overall runtime, being drastically overshadowed by both the pre-computation time, $S_{precomp}/R_{precomp}$,

and the cross-referencing time, $S_{crossref}/R_{crossref}$.

An instance of precomputing a subgraph and decomposing a full graph is functionally the same — both are full ZX-calculus reductions of ZX-diagrams. This explains why $R_{precomp} = R_{decomp}$. Furthermore, it is understandable that the cross-referencing calculations can be computed much more rapidly than decomposing and precomputing as these calculations are very simple, compared to full ZX-calculus decompositions and reductions.

Estimating the runtimes in this way is justified in that the results shown in this chapter aim to highlight *scales* and *trends*, rather than exact numerical runtimes (which at any rate would vary with hardware). Notably, these runtime rates are rather consistent (certainly with regard to order of magnitude). Indeed, the results presented are plotted on logarithmic scales and so the variance due to the uncertainty in the runtime rates above would not make a noticeable difference. This is especially true given that these small uncertainties would be negligible compared to the existing magnitudes-wide error margins (i.e. in figure 4.5b) due to the variance among the different ZX-diagrams.

As each cross-reference calculation can be computed much more quickly than each precomputation calculation, the program can in fact aim to balance the projected *runtimes* of these two parts, rather than simply their number of calculations. This step improves efficiency by ensuring neither part dominates the overall runtime. Lastly, note that all runtime results shown in this chapter quote the projected runtime *unless* this is below 100 seconds, in which case the real runtime is computed and measured. This ensures that speedy results - where there is more fluctuation and minor contributions to runtime can no longer be assumed to be negligible - are also accurate.

As a brief note on the memory overhead for the ZX-Partitioner, this scales as

$\max(\text{minpair}(H_i))$ for $i \in \{0, 1, \dots, k-2\}$ (see section 4.4). What this means is that, while this can result in Gigabytes of memory overhead, at scales beyond this the runtime would already become infeasible. In other words, assuming Gigabytes to be the upper-bound of what would be feasible, the memory overhead tends not to become infeasible before the runtime would - so this is not a limiting factor for the method.

Lastly, as suggested, the use of GPU parallelism here improves the rate, $R_{precomp}$, at which the unique subgraph scalars can be pre-computed. As per equation 4.59, this in turn is liable to reduce the total runtime for strong simulation. Given the hardware specified in section 4.6, this produced a measured improvement of factor 12 to the averaged pre-computation rate:

$$R_{precomp}^{gpu} = 21,400 \pm 13,300 \text{ calcs/s} \quad (4.62)$$

Similarly, the use of GPU parallelism when pairwise regrouping, as described in listing 4.1, provides a factor 3 improvement to the rate at which this step may be computed:

$$R_{crossref}^{gpu} = 412,000 \pm 145,000 \text{ calcs/s} \quad (4.63)$$

By balancing the projected *runtimes*, $\frac{S_{precomp}}{R_{precomp}}$ and $\frac{S_{crossref}}{R_{crossref}}$ per equation 4.59, rather than simply the forecast *number of calculations*, $S_{precomp}$ and $S_{crossref}$, a more practically significant and fine-tuned balance can be achieved. This ensures neither component dominates the runtime and that the balance between stabiliser decomposition and tensor contraction is optimally chosen as to minimise the total runtime of each.

4.7 Results

To benchmark the effectiveness of the ZX-Partitioner, its projected runtimes for fully reducing random Clifford+T ZX-diagrams was compared to that of directly reducing them via stabiliser decompositions (Kissinger et al., 2022) (with no partitioning). The same random dataset was also benchmarked for a *naïve* partitioning method which k -partitions (for its own self-determined optimal k) but does not apply the techniques outlined in this chapter. In particular, n -qubit circuits (of various n) were constructed by randomly placing gates of the set $\{T, S, HSH, CNOT\}$ with equal probability, up to the count of d gates (which is labelled its *depth*). (This is the `generate.cliffordT` function of PyZX.) The circuits were then plugged with $\langle + |^{\otimes n}$ and $| + \rangle^{\otimes n}$ to turn them into scalar diagrams, before finally they underwent an initial round of Clifford simplification. The resulting ZX-diagram, in each case, was taken as its initial state for the benchmarks, with the goal of each method being to fully reduce it to a scalar.

Figure 4.4 shows the \log_2 of the projected runtime results (in seconds) for each method on the random dataset of diagrams, varying in depth and number of qubits. The scale is \log_2 such that, approximately speaking, 0 represents a second, 6 a minute, and 12 an hour. 16, meanwhile, represents just over 18 hours, which is taken as the rough upper limit of what is computationally feasible. Thus, in these heatmaps, black denotes trivial cases while white denotes practically in-computable cases. The coloured region in between then represents the region of interest, which may be referred to as the *frontier*.

Note that for each circuit size, 10 random samples were computed, with the average result shown. The first observation one might make is that the ZX-Partitioner never performs meaningfully slower than the direct decomposition approach. This

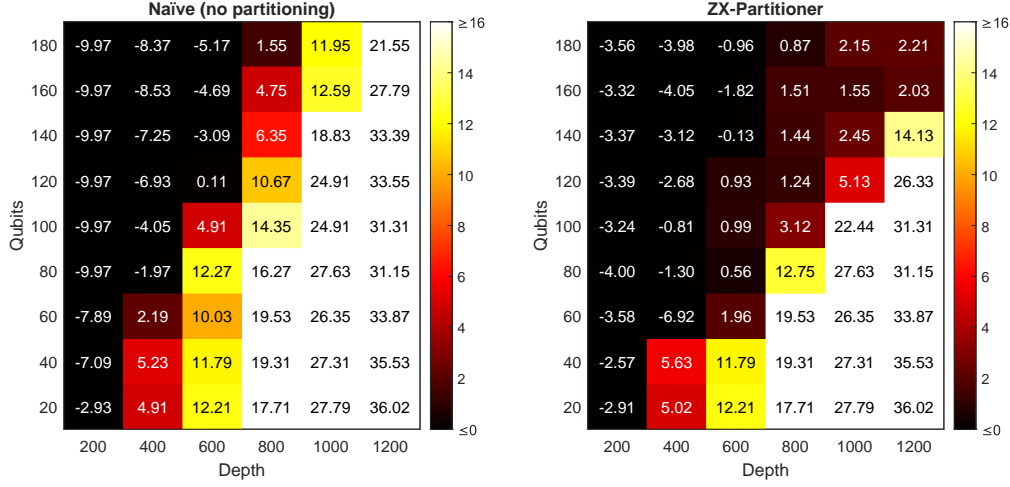


Figure 4.4: The average \log_2 runtimes (in seconds) for fully reducing a ZX-diagram via (left) the direct decomposition approach and (right) the smart partitioning approach, for uniformly random Clifford+T circuits of various depths and qubit counts.

is because the latter can be seen as a special case of the former, whereby the most optimal partition count is $k = 1$ (that is, where any amount of partitioning would result in worse performance and so the reduction proceeds without any). More significantly, the figure shows that, for certain sizes of circuits, the ZX-Partitioner method outperforms the naïve approach by many orders of magnitude.

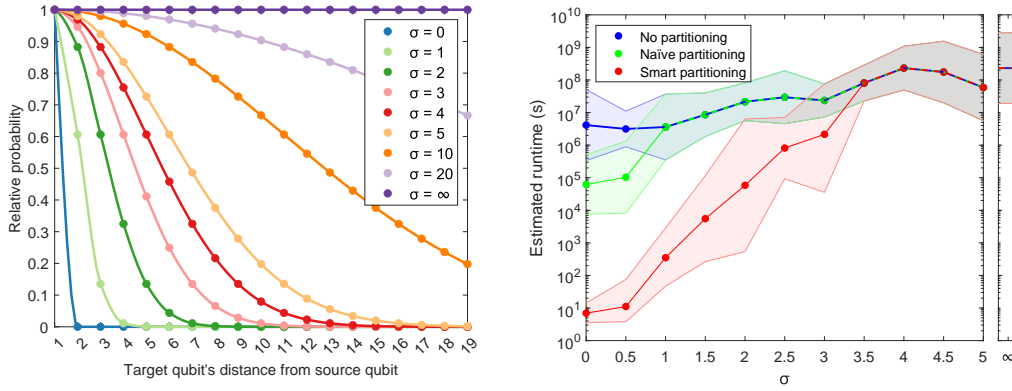
In practice, the ZX-Partitioner is most effective for many-qubit circuits of relatively low depth (albeit higher depths than the naïve approach), as well as few-qubit ($\lesssim 10$ qubits) circuits of any depth. In fact, in the latter case, if initial Clifford simplification were avoided and partitions were enforced along qubit-lines, then it would always be possible to partition an n -qubit circuit into arbitrarily many parts, connected linearly in a chain, where each part connects to the next via n edges. Consequently, with just slight modification, the smart partitioning approach could always achieve a runtime proportional to $O(2^{2n})$. However, few-qubit circuits are already known to be efficiently simulable (by computing the

state vector (Jamadagni et al., 2024) or via tensor contraction (Markov & Shi, 2008)). This leaves the more interesting case of shallow many-qubit circuits. It is easy to understand why these circuits are also particularly effective for the smart partitioner approach, as ‘depth’ in this context refers to the total number of gates. Hence, when the ratio of the depth to the number of qubits is low, this describes circuits with few gates per qubit, and hence few CNOTs connecting these qubits, meaning few cuts would likely be needed to partition along these lines.

Beyond these cases, the ZX-Partitioner appears to offer no improvement versus direct decomposition. However, recall that this dataset was generated completely randomly, and so it is understandable that the frequency of good vertex cuts for partitioning shrank as the overall size of the graphs grew. In more realistic circuits, one would expect more inherent structure and, as such, a less sporadic placement of CNOTs. Indeed, most physical implementations constrain gates to nearby qubits (Markov & Shi, 2008). To try to model this with a new randomly generated dataset, a slight modification is needed in how the CNOTs are placed. Whereas previously both ends of the CNOT were placed on different random qubits, the new dataset instead places one end of the CNOT on a random qubit and then randomly decides the qubit of its other end according to a non-uniform probability distribution, to favour nearer qubits over further ones. Specifically, when deciding where to place the target qubit of the CNOT, the probabilities of the qubits are weighed according to a normal distribution about the control qubit, such that the probability of a CNOT spanning $\Delta q \geq 1$ qubits is given by:

$$P(\Delta q) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\Delta q - 1)^2}{2\sigma^2}}$$

This is derived from the general form of the normal distribution function, where σ denotes the standard deviation (and hence σ^2 denotes the variance). Figure 4.5a



(a) The probability amplitudes for a CNOT spreading Δq qubits, according to a normal distribution with a standard deviation, σ .

(b) The average projected runtimes for strongly simulating Clifford+T circuits of 30 qubits and a depth (gate count) of 1,000, using three different methods.

Figure 4.5: Random Clifford+T circuits were generated, with the spread of each CNOT decided probabilistically according to a normal distribution. By adjusting the variance of this distribution (left), the runtime for strongly classically simulating the circuits (right) is also affected.

shows what this distribution looks like for various values of σ . Note that these have each been scaled (un-normalised) to show the probabilities relative to that of $P(\Delta q = 1)$. This way they are all clearly readable within the same plot. For instance, when $\sigma = 3$ the span of a CNOT (i.e. the distance between its control and target) is roughly 0.6 times as likely to be exactly 4 qubits as it is to be exactly 1 qubit. Moreover, when $\sigma = 0$, this means that CNOTs always connect to their nearest neighbouring qubit (either immediately above or immediately below, with equal probability). Meanwhile, at the other extreme, when $\sigma = \infty$, the target of each CNOT will be placed on any of the qubits in the circuit with uniformly equal probability (as in the experiments of figure 4.4).

Given this modification, consider — as an illustrative example — randomly generated circuits of 30 qubits and depth 1,000. On such circuits, this experiment was repeated for various values of σ , in each case taking the averaged *log* runtime

over 10 repeats. The direct decomposition (i.e. no partitioning) method, as well as the naïve partitioning method, were also tested against the same dataset. The results are shown in figure 4.5b, with error bars given by the standard deviation of the \log runtimes over the 10 repeats for each σ .

From this figure one will immediately observe that the effectiveness of the smart partitioning method is heavily impacted by σ . For circuits of this particular size, one may notice that the method generally outperforms both the ‘no partitioning’ and ‘naïve partitioning’ approaches (often by many orders of magnitude) when $\sigma \lesssim 3$. Indeed, as lower values of σ are used to generate the random circuits, this improvement becomes ever more drastic, though even for very small σ , the smart partitioner’s runtime doesn’t fall much below a second, despite what would be predicted by estimating the runtimes from the number of precomputing and cross-referencing calculations. This is because, in these very speedy cases, the overhead runtime from the partitioning function itself (which can take up to a few seconds in the most extreme cases) is no longer negligible.

Moreover, the direct decomposition (‘no partitioning’) approach is relatively consistent as σ is varied (as compared to the partitioning methods). This is because the complexity of this approach depends primarily upon the initial T-count, which itself is influenced by the distribution of CNOTs in only very roundabout ways. Additionally, neither partitioning method ever performs meaningfully slower than the naïve approach because, as noted earlier, ‘no partitioning’ is essentially the $k = 1$ special case of partitioning

Furthermore, one may see here that once the partitioning method has been capped by the naïve approach (that is, when no partitioning becomes optimal) it tends to remain so as σ is further increased. Lastly, as $\sigma \rightarrow \infty$ the methods each reach their terminal runtimes. In this case — because $k = 1$ is deemed optimal —

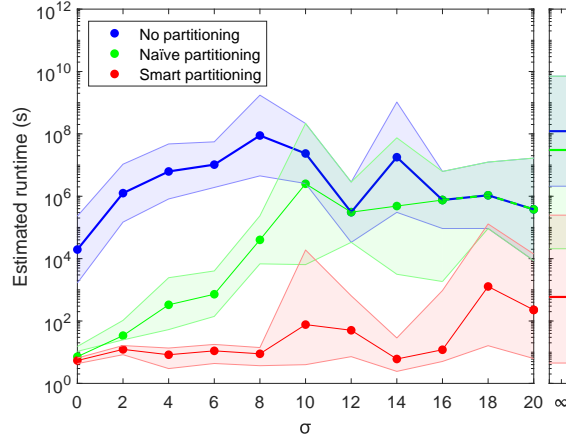


Figure 4.6: The average projected runtimes for strongly simulating Clifford+T circuits of 110 qubits and a depth (gate count) of 1,000, using three different methods.

they each share the same terminal runtime, namely $10^{8.36 \pm 1.09}$ seconds (which is consistent with what was observed in figure 4.4). However, as evidenced by figure 4.4, there are circuit sizes for which even $\sigma = \infty$ leads to the smart partitioner method reigning supreme, and in such cases — if plotted against σ like figure 4.5b — one would observe that the smart partitioning method always remains below the others. One such example is shown in figure 4.6.

Lastly, the figure 4.4 experiments were repeated for randomly generated circuits with $\sigma = 2$ (as opposed to $\sigma = \infty$). These results are shown in figure 4.7. Under these conditions, the naïve approach has scarcely changed, yet the ZX-Partitioner shows significantly reduced runtimes, with a far shallower frontier. Clearly, therefore, CNOTs with a more localised spread (i.e. a lower σ) lead to circuits which are much more partitionable and hence even more suitable for such methods. Indeed, given low σ , one may observe that for all bar the most unfavourably sized circuits, the partitioner method offers orders of magnitudes reduction to the runtime versus the naïve alternative. (Furthermore, section 4.8

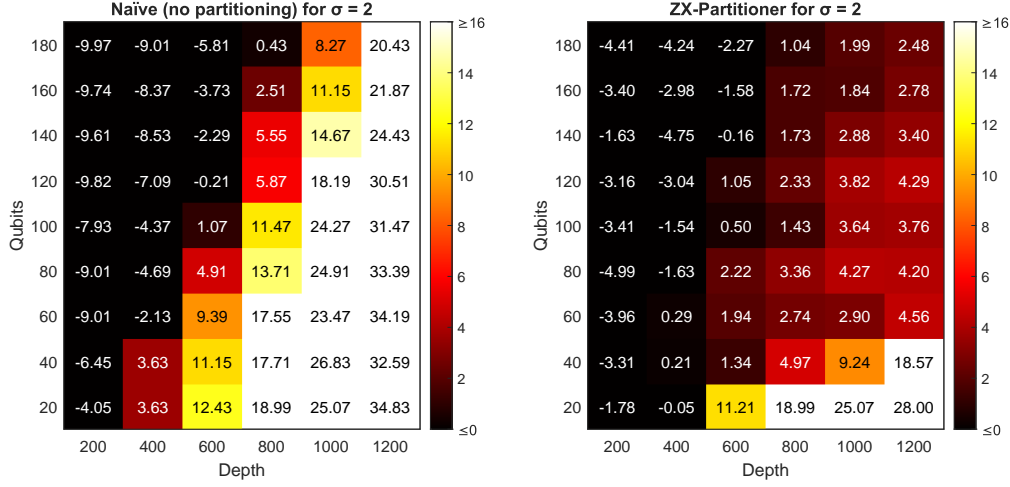


Figure 4.7: The average \log_2 runtimes (in seconds) for fully reducing a ZX-diagram via (left) the direct decomposition approach and (right) the smart partitioning approach, for random Clifford+T circuits of various depths and qubit counts, with the spread of each CNOT given by a normal distribution with $\sigma = 2$.

discusses how these results compare to those achieved by tensor contraction and consider a slightly modified type of randomly generated circuit which is perhaps more realistic.)

4.8 Tensor Contraction and Compound Circuits

The smart partitioner method detailed in this chapter combines stabiliser decomposition with a tensor contraction approach. As such, it is helpful to also consider how it compares to such tensor contraction approaches (Markov & Shi, 2008; Brennan et al., 2021) to strong classical simulation. Both have a memory overhead and runtime complexity that grow exponentially with the interconnectedness (or *treewidth*) of the circuit. Indeed, for the experiments run in this chapter, many of the cases that were particularly favourable to the smart partitioner approach (such as shallow circuits or those with especially low σ) were also effectively

simulable with the tensor contraction method.

Despite this, it is not strictly true that all circuits for which the smart partitioner method is effective could also be effectively simulated via tensor contraction alone. In fact, it is very easy to design example cases which showcase this point. As a very simple example, consider a set of subgraphs, G_1, \dots, G_5 , which are each *internally* very highly interconnected such that each individually is beyond the scope of tensor contraction but within the scope of stabiliser state decomposition. Now suppose these subgraphs are connected *to one another* in a relatively inexpensive way (that is, with relatively few edges).

As each subgraph is individually beyond the scope of tensor contraction, it follows that whole is likewise. Nevertheless, the smart partitioner method could very effectively (and at a relatively small computational cost) partition the graph into its 5 locally dense subgraphs and reduce each using stabiliser decomposition, before cross-referencing the results to attain the final amplitude. Indeed, on randomly generated examples similar to this, experimental results verified that tensor contraction (using the *Quimb* (Gray, 2018) Python library) would fail (due to exceeding a reasonable runtime limit or 128GB of memory overhead) while the smart partitioner would complete within seconds and requiring (in some cases) only a matter of bytes in memory overhead.

So, while there is a notable overlap in the applicability of these two methods on the types of circuits used in the experiments presented in this chapter, this essentially is a consequence of the means by which the random circuits were generated. With slight modification, one can generate a similar class of random circuits, which have non-uniform CNOT spreads and are realistically justified and which are (generally) effectively simulated via the smart partitioner but not tensor contraction.

Specifically, one can randomly generate k distinct Clifford+T circuits of q qubits (with uniform CNOT spreads, i.e. $\sigma = \infty$). These circuits may be vertically composed and some number, n , of additional CNOTs may be inserted which each connect *between* some pair of the sub-circuits. For each of these external CNOTs, when deciding how far away the target sub-circuit should be from the source, one can - as before - use a normal distribution (albeit acting on a sub-circuit by sub-circuit level rather than a qubit by qubit level). Generating circuits in this way leads to structures like that of figure 4.8.

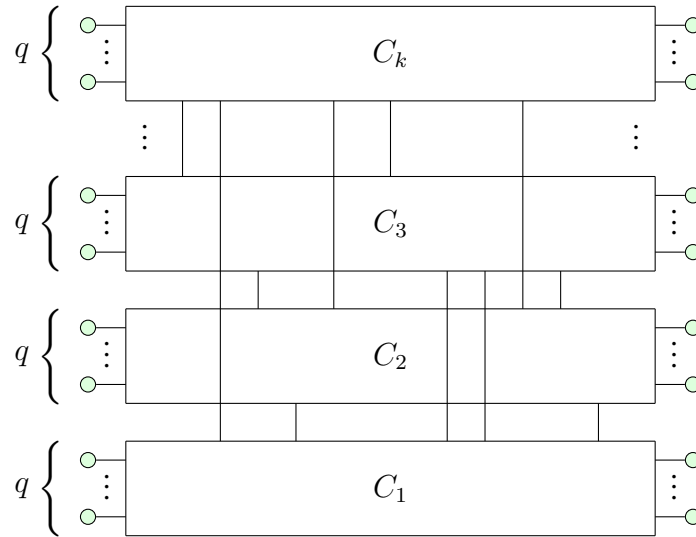


Figure 4.8: The structure of a compound circuit.

These ‘*compound circuits*’ manifest highly interconnected local cliques, which in turn are connected to one another by only a relatively modest number of edges. Moreover, they arguably offer fairly realistic examples of circuit structures, being composed of smaller independent subroutines which relay some information to one another.

While it is difficult to fairly quantify such results (as such circuits can be made as generously or ungenerously to these aims as desired), in preliminary experiments, it was observed that - generally speaking - such circuits are practically unsim-

uitable for both tensor contraction and direct decomposition, yet are effectively simulated with the smart partitioning approach presented in this chapter.

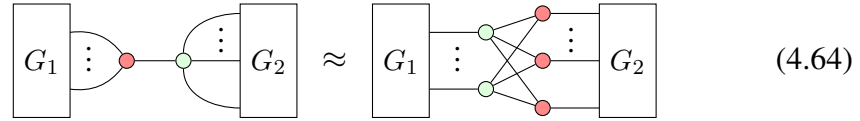
4.9 Improving Partitionability

Presented in this chapter thus far are a number of optimisations to the calculations involved in a partitioning-aided method of quantum circuit simulation. However, the partitioning function itself remains essentially unaltered from the graph theory literature. Notably, the partitioner treats ZX-diagrams as generic graphs, without any regard for how they may be transformed via the rewriting rules. Addressing this issue, the rules of ZX-calculus could be utilised to optimise the ZX-diagrams in such a way as to improve partitionability. This has the potential to yield drastic further reductions to the runtime. Indeed, this could be very interesting and promising area of research in its own right, being a twist on the usual simplification strategies which aim to exclusively minimise, for instance, T-count (usually at the expense of increased edge connectivity).

While this topic has not yet been extensively researched within the scope of this thesis and remains for future work, there are nevertheless some ideas and considerations worth expressing here. For instance, recall that the simplification strategy used for reducing scalar ZX-diagrams, as described in section 2.2.2, makes significant use of both pivoting and local complementation (see figure 2.3). These rules, whenever applicable, are applied left to right to aid in spider (and hence T-count) minimisation. However, notice that this comes at the cost of greatly increasing the edge connectivity among the remaining spiders, which may serve to considerably hinder partitionability. Consequently, it may be advisable to be more discriminatory when deciding when and where to apply these rules, or even to apply them in reverse (right to left) where it may be appropriate. For instance, in many cases

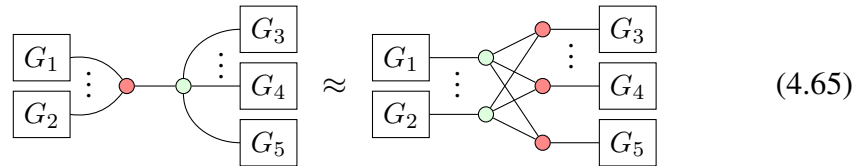
it will likely be worthwhile to *un-gadgetise* the phase gadgets after full Clifford simplification.

Likewise, consider the bialgebra rule (see figure 2.2). Appropriately applying this rule in reverse (right to left) could also be very helpful in aiding partitioning. Suppose the leftward edges connect to a subgraph, G_1 , and the rightward edges to a subgraph, G_2 , which are otherwise unconnected:



$$(4.64)$$

Assume there are n leftward edges entering G_1 and m rightward edges entering G_2 . By applying the bialgebra rule in reverse one can reduce the number of vertex cuts required to separate these two subgraphs from $\min(n, m)$ down to just 1. Similarly, imagine each of these $n + m$ outgoing edges were connected to its own independent (and otherwise disconnected) subgraph:



$$(4.65)$$

In this case, applying bialgebra right to left would reduce the number of cuts required to fully disconnect all these subgraphs from $\min(n, m)$ down to just 2.

Moreover, from local complementation and the cutting decomposition (plus the derivable rule whereby two parallel Hadamard edges between a pair of like-coloured spiders may cancel out (van de Wetering, 2020)), one may derive a new rule which allows one to toggle the (Hadamard) edge connectivity among any set of $n \geq 2$ like-coloured spiders at the cost of one cut, as in the following example:

$$\begin{array}{c}
 \begin{array}{ccc}
 \alpha_1 & & \alpha_4 \\
 & \diagdown & / \\
 & \alpha_2 & \alpha_3 \\
 & / & \diagdown
 \end{array}
 =
 \begin{array}{ccc}
 \alpha_1 & & \alpha_4 \\
 & \diagdown & / \\
 & \alpha_2 & \alpha_3 \\
 & / & \diagdown
 \end{array}
 \end{array}
 \quad (4.66)$$

$$\approx
 \begin{array}{ccc}
 \alpha_1 + \frac{\pi}{2} & & \alpha_4 + \frac{\pi}{2} \\
 & \diagdown & / \\
 & \frac{\pi}{2} & \\
 & / & \diagdown \\
 \alpha_2 + \frac{\pi}{2} & & \alpha_3 + \frac{\pi}{2}
 \end{array}
 \approx
 \sum_a^{\{0,1\}}
 \begin{array}{ccc}
 \alpha_1 + \frac{\pi}{2} + a\pi & & \alpha_4 + \frac{\pi}{2} + a\pi \\
 & \diagdown & / \\
 & & \\
 & / & \diagdown \\
 \alpha_2 + \frac{\pi}{2} + a\pi & & \alpha_3 + \frac{\pi}{2} + a\pi
 \end{array}$$

Notice that in the initial diagram (left-hand side) of the above example, one could alternatively have chosen to toggle the edge connectivity among the set $\{\alpha_1, \alpha_2, \alpha_3\}$ or $\{\alpha_1, \alpha_2, \alpha_4\}$, or even α_1, α_4 or α_2, α_3 , etc. Each of these options would have represented fully connected cliques and so applying the new rule in any of these cases would have removed edges without introducing any new ones. Nevertheless, the best of these options would have only partitioned the diagram into 2 disconnected parts, whereas the incomplete (but near) clique set of $\alpha_1, \alpha_2, \alpha_3, \alpha_4$, to which the rule was applied in equation 4.66, enabled partitioning into 3 disconnected parts.

When deciding when and where to apply this rule, it is not always obvious, particularly among larger graphs with lots of connected cliques and near-cliques, what the best set of spiders to select in each case is. The following example illustrates this point:

$$\begin{array}{ccc}
 \begin{array}{c} G_1 \\ | \\ \alpha_1 \\ / \backslash \\ G_5 \quad G_2 \\ | \quad | \\ \alpha_5 \quad \alpha_2 \\ \backslash / \\ \alpha_4 \quad \alpha_3 \\ | \quad | \\ G_4 \quad G_3 \end{array} & \approx \sum_a^{\{0,1\}} & \begin{array}{c} G_1 \\ | \\ \alpha_1 + \frac{\pi}{2} + a\pi \\ / \backslash \\ G_5 \quad G_2 \\ | \quad | \\ \alpha_5 + \frac{\pi}{2} + a\pi \quad \alpha_2 + \frac{\pi}{2} + a\pi \\ \backslash / \\ \alpha_4 \quad \alpha_3 + \frac{\pi}{2} + a\pi \\ | \quad | \\ G_4 \quad G_3 \end{array} \\
 & & (4.67) \\
 & & \begin{array}{c} G_1 \\ | \\ \alpha_1 + \frac{\pi}{2} + a\pi \\ / \backslash \\ G_5 \quad G_2 \\ | \quad | \\ \alpha_5 + \pi + a\pi + b\pi \quad \alpha_2 + \frac{\pi}{2} + a\pi \\ \backslash / \\ \alpha_4 + \frac{\pi}{2} + b\pi \quad \alpha_3 + \pi + a\pi + b\pi \\ | \quad | \\ G_4 \quad G_3 \end{array}
 \end{array}$$

Here, there are 5 otherwise disconnected subgraphs, G_1, \dots, G_5 , which meet among these 5 spiders, $\alpha_1, \dots, \alpha_5$. There are many different ways in which the new rule may be applied to fully disconnect these subgraphs, with option for the cheapest approach (costing just 2 cuts) shown here. Yet, devising an algorithm to determine the optimal applications of this rule (particularly for cases too large and complex to deduce by inspection) is something which remains for future research.

Due in part to the partitioning function described in section 2.4.2 being a heuristic method, the ‘*partitionability*’ of a ZX-diagram (that is, how effectively it may be partitioned into $k \geq 2$ subgraphs such that the overall runtime for classical simulation is minimised) is difficult to describe and formulate. It evidently depends upon such factors as edge connectivity, but does so in very complex, and often indirect, ways. Simple metrics such as the overall number of edges may, generally, be helpful indicators of the partitionability of a ZX-diagram, but even this, as shown above, can be difficult to optimise for, and indeed it is possible for a

ZX-diagram to become *less* partitionable when the number of edges is reduced, if the *locations* of these edges become undesirable.

While it can be difficult to know *how* to effectively optimise for partitionability, what is known is the set of actions one may take, namely the rewriting rules (in both directions). Furthermore, while difficult to formulate, the partitionability is, as per section 4.6, measurable and quantifiable. Given these observations, the problem of optimising for partitionability lends itself well to reinforcement learning. Such work remains for future research, with substantial potential for improving the results even further.

4.10 Conclusions

The existing literature (Codsì, 2022) on partitioning ZX-diagrams for classical simulation, as outlined in section 2.4.3, essentially amounts to three observations:

- A scalar ZX-diagram composed of two or more disjoint subgraphs may be simulated more efficiently by computing each subgraph independently (lemma 4).
- A ZX-diagram may be partitioned by inducing c vertex cuts, at the computational cost exponential in c (lemma 5).
- Existing state of the art hypergraph partitioning methods (Schlag, 2020) may be employed to find a heuristically optimal partitioning of a ZX-diagram (see definition 23).

The present chapter furthers this literature — significantly improving both the efficiency and applicability for a partitioning-based method of classical simulation via ZX-calculus — via a handful of new techniques and optimisations, namely:

1. applying the GPU parallelism work of chapter 3 (as well as GPU parallelism for the tensor contraction calculations), and
2. precomputing the unique sub-scalars to avoid redundancy in recalculating them,
3. regrouping the partitioned segments' scalar profiles pairwise in a tensor contraction like approach, and
4. optimally k -partitioning the ZX-diagrams and optimally balancing the computational workload of stabiliser decomposition and tensor contraction.

Functionally, this method acts rather like a hybrid of a stabiliser decomposition approach and a tensor contraction approach, reaping the benefits of both and largely evading the limitations of either. Essentially, stabiliser decomposition is employed within dense local cliques, with a tensor contraction like technique then utilised to collapse these together. As such, dense local cliques, which are each high in interconnectedness but contain only a fraction of the diagram's total T-count, are reduced using stabiliser decomposition, with a complexity scaling exponentially with this reduced local T-count. Conversely, these cliques themselves are connected to one another with considerably fewer edges and are reduced via a tensor contraction like approach whose complexity scales exponentially with this modest cross-clique edge count.

Benchmarked against the direct decomposition (no partitioning) approach, as well as a naïve partitioning approach, the results demonstrate a runtime increase of orders of magnitude for random circuits under realistic (low σ) conditions. Furthermore, its effectiveness is shown to vary with the interconnectedness of the circuits, and promising future research to further optimise for this is proposed in section 4.9.

5 | Procedurally Optimised ZX-Diagram Cutting

A broad assumption among the ZX-calculus literature relating to classical simulation is that more efficient results will be achieved by utilising more efficient (lower α) stabiliser decompositions. Consequently, much of the research on this topic has been dominated by work (Kissinger et al., 2022; Koch et al., 2023; Codsi, 2022; Laakkonen, 2022) focused on discovering new decompositions of ever lower α .

Conversely, the present chapter seeks to challenge this assumption by demonstrating how apparently weaker (higher α) decompositions can, in fact, lead to more efficient overall results, when applied thoughtfully. This observation was implicit in the work of (Codsi, 2022) but is more formally and substantially addressed here.

Specifically, this chapter identifies a general pattern, common among Clifford+T circuits, to which the cutting decomposition enables significant T-count reduction. This is paired with a weighting heuristic upon the gates, together with a means of propagating these weights to take into account the broader neighbourhood, such that (heuristically) optimal applications of the decomposition may be determined. In short, this approach procedurally designs a sequence of vertex cuts optimised for the particular circuit provided.

Ultimately, the method is benchmarked against the more conventional ZX-calculus based approaches, outperforming the state of the art when applied to structured circuits, with a practical effective α efficiency considerably below that of the alternative methods.

This chapter is based upon the work introduced in (Sutcliffe & Kissinger, 2024b).

5.1 Efficient Graph Cutting

Most of the ZX-calculus literature on stabiliser decomposition approaches to classical simulation (Kissinger et al., 2022; Koch et al., 2023; Codsi, 2022; Laakkonen, 2022) has aimed at discovering new, more efficient, decompositions of lower α in an effort to reduce the rate at which the number of stabiliser terms grows with the initial T-count. As such, ever more elaborate decompositions have been employed to this problem, including those of section 2.3.4.

In contrast, and perhaps contrary to intuition, this chapter relies exclusively upon the simplest and most naïve decomposition of all, namely the cutting decomposition presented in equation 2.68. Unlike the more sophisticated decompositions in the literature, which achieve α values as low as 0.396 (Qassim et al., 2021) (or even lower when including the asymptotic α of known *partial* decompositions (Kissinger et al., 2022)), the cutting decomposition, applied to a T-like spider, achieves a markedly meagre efficiency of $\alpha = 1$.

Despite its ostensibly poor α , this unassuming decomposition in its simplicity possesses a few key features that render it rather powerful in practice, namely:

- It has **perfect parametric symmetry**, meaning every time it is applied its two branching terms can be expressed as one parametric term, effectively allowing simplification and reasoning (and, to some extent, evaluation) across all branches simultaneously.
- It is **highly locally partitioning**, in that whenever it is applied to an n -degree spider, it locally partitions n -ways, making it effective at separating ZX-diagrams into disjointed parts.
- It is **universally applicable**, meaning it does not depend on some specific

structure but rather can be applied anywhere there is a spider (of any colour, edge-set, and phase).

- It has a **low stabiliser rank** of 2, meaning each application only increases the total term count by a factor of 2.

In addition to these points, the cutting decomposition is also dynamic, being applicable to both *Z*-spiders *and* *X*-spider (and indeed even edges), as well as a *total* (as opposed to partial) stabiliser decomposition. Lastly, it is also very neat, which makes its behaviour easier to track as compared to decompositions which introduce many sporadic edge connections. On the whole, it is a very modest, and therefore often overlooked, decomposition whose beauty lies in its simplicity.

The hypothesis of this chapter is thus to consider how well-motivated applications of this decomposition can in fact lead to very efficient results, with a lower effective α than would be achieved with the apparently more powerful decompositions.

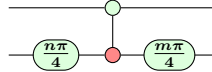
5.1.1 Slicing Spider Sandwiches

When a T-decomposition is applied, there is an initial set of T-spiders that are removed and replaced with stabiliser terms. This is quantified by the α efficiency of the decomposition. However, what this fails to take into account is the extent to which further T-spiders may be removed thanks to ZX-calculus simplification facilitated by the decomposition. Measuring the α_{eff} after such simplification provides a much more practical quantity of the decomposition's efficiency, though this will vary from circuit to circuit and is not determinable in advance of its application.

Nevertheless, by analysing a given circuit at a broader scale than that to which the prospective decomposition is to be applied, one can gauge whether significant simplification, leading to notable T-count reduction, is likely to result. Searching

for particular patterns and structures is one way to approach this.

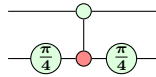
Definition 31. A *spider sandwich* describes a pattern within a ZX-diagram consisting of two like-coloured T-like spiders sat either side of the opposite coloured end of a CNOT, like so:



where n and m are odd integers and all spider colours may be inverted.

Consider, for instance, a pair of T-like spiders sandwiched between a CNOT, as pictured in definition 31. Such a pattern may be labelled a ‘*T-CNOT-T sandwich*’ or, rather grotesquely, a ‘*spider sandwich*’. While cutting either T-spider here would only reduce the T-count by one, it is in fact possible to remove both of the T-spiders by cutting one of the Clifford spiders, as addressed in lemma 28.

Lemma 28. A *spider sandwich*, such as follows:



may be decomposed into two fully Clifford terms via one cut.

Proof. Cutting one end of the CNOT (the end matching in colour to the T-spiders) in such a structure produces two terms, like so:

$$\begin{array}{c} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \end{array} \approx \begin{array}{c} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \end{array} + \begin{array}{c} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \end{array} \quad (5.1)$$

Using the rewriting rules of ZX-calculus, each of these terms is reducible to Clif-

ford:

$$\begin{array}{c} \text{---} \bullet \text{---} \quad \bullet \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \bullet \text{---} \circ_{\pi/4} \text{---} \end{array} = \begin{array}{c} \text{---} \bullet \text{---} \quad \bullet \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \bullet \text{---} \circ_{\pi/4} \text{---} \end{array} = \begin{array}{c} \text{---} \bullet \text{---} \quad \bullet \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \circ_{\pi/4} \text{---} \end{array} = \begin{array}{c} \text{---} \bullet \text{---} \quad \bullet \text{---} \\ \text{---} \circ_{\pi/2} \text{---} \end{array} \quad (5.2)$$

and:

$$\begin{array}{c} \text{---} \pi \text{---} \quad \pi \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \bullet \text{---} \circ_{\pi/4} \text{---} \end{array} = \begin{array}{c} \text{---} \pi \text{---} \quad \pi \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \pi \text{---} \circ_{\pi/4} \text{---} \end{array} = e^{i\pi/4} \begin{array}{c} \text{---} \pi \text{---} \quad \pi \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \pi \text{---} \pi \text{---} \end{array} = e^{i\pi/4} \begin{array}{c} \text{---} \pi \text{---} \quad \pi \text{---} \\ \text{---} \pi \text{---} \end{array} \quad (5.3)$$

□

Thus, despite the cut itself being applied to a Clifford spider and hence achieving an ineffectual $\alpha = \infty$, after simplification this in fact leads to $\alpha_{\text{eff}} = 0.5$. Indeed, a slight variation of this would see a T-like spider being cut and achieve $\alpha_{\text{eff}} = \frac{1}{3}$, such as follows:

$$\begin{array}{c} \text{---} \circ_{\pi/4} \text{---} \\ \text{---} \circ_{\pi/4} \text{---} \bullet \text{---} \circ_{\pi/4} \text{---} \end{array} \approx \begin{array}{c} \text{---} \bullet \text{---} \quad \bullet \text{---} \\ \text{---} \circ_{\pi/2} \text{---} \end{array} + e^{i\pi/2} \begin{array}{c} \text{---} \pi \text{---} \quad \pi \text{---} \\ \text{---} \pi \text{---} \end{array} \quad (5.4)$$

5.1.2 CNOT Grouping

More generally, it is possible for any number of such spider sandwiches to be aligned such that their CNOTs may fuse and be cut as one. In such cases, each pair of T-like spiders may fuse to Clifford as above, though still at the cost of a single collective cut. This is detailed in lemma 29.

Lemma 29. *A set of adjacent spider sandwiches, each with their CNOT's controls*

sharing a common qubit, may reduce to two Clifford terms:

$$\begin{aligned}
 & \dots \text{---} \overset{\beta_1}{\circ} \text{---} \overset{\beta_2}{\circ} \text{---} \dots \\
 & \dots \text{---} \overset{n_1^1 \frac{\pi}{4}}{\circ} \text{---} \overset{a_1 \pi}{\circ} \text{---} \overset{n_1^2 \frac{\pi}{4}}{\circ} \text{---} \dots \\
 & \dots \text{---} \overset{n_2^1 \frac{\pi}{4}}{\circ} \text{---} \overset{a_2 \pi}{\circ} \text{---} \overset{n_2^2 \frac{\pi}{4}}{\circ} \text{---} \dots \\
 & \approx A \dots \overset{m_1 \frac{\pi}{4}}{\circ} \text{---} \overset{a_1 \pi}{\circ} \text{---} \dots \\
 & \dots \overset{m_2 \frac{\pi}{4}}{\circ} \text{---} \overset{a_2 \pi}{\circ} \text{---} \dots \\
 & + B \dots \overset{m'_1 \frac{\pi}{4}}{\circ} \text{---} \overset{(a_1 + 1) \pi}{\circ} \text{---} \dots \\
 & \dots \overset{m'_2 \frac{\pi}{4}}{\circ} \text{---} \overset{(a_2 + 1) \pi}{\circ} \text{---} \dots
 \end{aligned} \tag{5.5}$$

where:

$$\begin{aligned}
 n_i^j &\in \mathbb{Z} \text{ is odd } \forall i, j \\
 a_i &\in \mathbb{B} \forall i \\
 \beta_i &\in \mathbb{R} \forall i
 \end{aligned} \tag{5.6}$$

and:

$$\begin{aligned}
 A &:= e^{i \frac{\pi}{4} (a_1 n_1^2 + a_2 n_2^2 + \dots)} \\
 B &:= e^{i \frac{\pi}{4} (n_1^2 - a_1 n_1^2 + n_2^2 - a_2 n_2^2 + \dots) + i(\beta_1 + \beta_2 + \dots)}
 \end{aligned} \tag{5.7}$$

and $\forall i$:

$$\begin{aligned}
 m_i &:= n_i^1 + n_i^2 - 2a_i n_i^2 \\
 m'_i &:= n_i^1 - n_i^2 + 2a_i n_i^2
 \end{aligned} \tag{5.8}$$

such that $m_i, m'_i \in \mathbb{Z}$ are odd $\forall i$ and hence $m_i \frac{\pi}{4}$ and $m'_i \frac{\pi}{4}$ are necessarily Clifford.

Given N such adjacent spider sandwiches, the T -count reduction that may be induced by a single cut is then $2N + b$, where:

$$b = \begin{cases} 0, & \text{if } \frac{4}{\pi} \sum_{i=1}^N \beta_i \text{ is even,} \\ 1, & \text{if } \frac{4}{\pi} \sum_{i=1}^N \beta_i \text{ is odd.} \end{cases} \tag{5.9}$$

Hence, such a cut, after simplification, achieves:

$$\alpha_{\text{eff}} = \frac{1}{2N + b} \quad (5.10)$$

A simple example follows, demonstrating how a pair of adjacent spider sandwiches may reduce in this way:

Example 16.

$$\begin{aligned}
 & \text{Diagram 1} = \text{Diagram 2} \\
 & \approx \text{Diagram 3} + \text{Diagram 4} \\
 & = \dots = \text{Diagram 5} + e^{\frac{i\pi}{2}} \text{Diagram 6}
 \end{aligned} \quad (5.11)$$

In this case, the T-count has reduced by 4, at the expense of one vertex cut (hence 2 Clifford terms). This corresponds to $\alpha_{\text{eff}} = 0.25$ (or indeed $\alpha_{\text{eff}} = 0.2$ had the cut spider instead been T-like).

Naturally, there is no upper bound for the number of adjacent spider sandwiches which could be reduced to two Clifford terms in this fashion, with $\alpha_{\text{eff}} \rightarrow 0$.

While not expressed in quite these terms, a similar concept was recognised in (Codsi, 2022), where T-count reductions of up to 286 were achieved via a single cut in practical SAT counting circuits (de Beaudrap et al., 2020; Berent,

5.1.3 Cutting in Tiered Structures

208

Indeed, there are a number of possible ways of reducing this circuit via the means outlined so far in this chapter.

For example, cutting vertices #4, #9, and #13 would allow 6 T-gates to reduce to Clifford. Alternatively, cutting #2, #6, #11, and #15 would reduce all 8 T-gates, at the cost of 4 cuts (hence $\alpha = 0.5$). But, the best solution here would be to firstly cut vertex #8 - even though this doesn't immediately allow any T-gates to reduce - as this then allows vertices #2, #6, #11, and #15 to fuse into one. Cutting this newly fused vertex then would allow all 8 T-gates to reduce. So, this solution would reduce the T-count entirely (all 8 T-gates) at the cost of just 2 cuts (hence $\alpha = 0.25$). Of course, this reasoning could be extended for higher tiers, where the optimal initial cuts are two, three, or more, steps away from any T-gates.

Similarly, there may be instances where *multiple* CNOTs are directly blocking some set of T-gates from fusing. In such cases, the likelihood that cutting all of those CNOTs would be worthwhile to reduce the T-gates they block will be determined by the ratio of the CNOTs to T-gates involved, as well as whether some of those CNOTs are considered worthwhile cuts in their own right, with regard to any other T-gates that they alone may be blocking. One may call such groups of spiders '*spousal*', with respect to the children spiders they are collectively blocking from reducing.

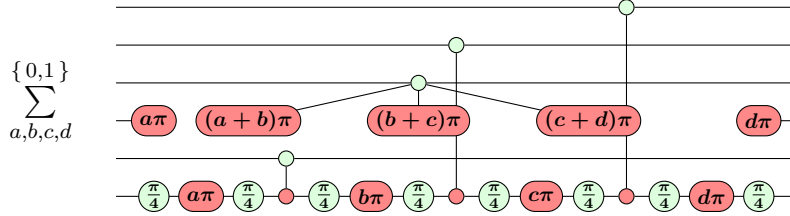
Evidently, therefore, selecting which vertices to cut, and indeed in which order, is a very intricate task (though the former is more important as suboptimality in the latter can be corrected for via a slight modification and parametric analysis, as detailed in the next subsection). Naïvely tackling this problem via an exhaustive, brute force, approach would require checking the reduction achieved by every possible combination of vertex cuts. This is obviously infeasible for large-scale graphs, as the time complexity scales exponentially with the number of vertices.

Consequently, a heuristic approach is desired. On this note, as has been shown here, simply prioritising vertices which are directly blocking the most number of T-like pairs from fusing is not generally optimal. Rather, it is preferential to look at the whole picture and determine the optimal cuts on higher tiers.

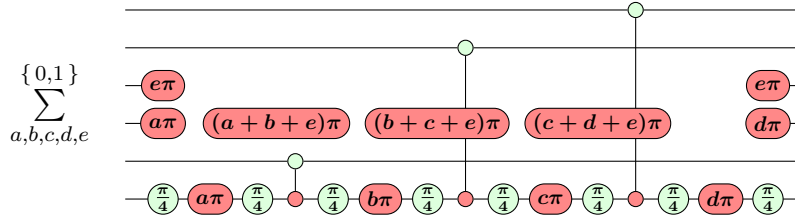
5.1.4 Cut Order Correction

It was shown in section 5.1.3 that determining the *order* in which to cut the vertices is apparently at least as important as determining *which* vertices to cut. In the example showcased there, it seemed necessary for the optimal solution to cut vertex #8 first, such that vertices #2, #6, #11, and #15 could then be fused and cut as one. It would appear that recognising the same vertices to cut but applying a different cut ordering (specifically cutting vertex 8 *last*) would achieve the same ends at the cost of 5 cuts rather than 2 (and hence a much less efficient 32 stabiliser terms rather than 4). However, with slight alteration (using the modified PyZX package of (Sutcliffe, 2024a)) to denote the cuts parametrically, followed by some simple parametric analysis, a suboptimal cut ordering can effectively be corrected to its more optimal arrangement at a negligible cost to the runtime.

Firstly, one may consider the parameterised formulation of the cutting decomposition, as presented in equation 3.75. Consider again the circuit shown in section 5.1.3, and imagine cutting first vertices #2, #6, #11, and #15. Writing this in the parameterised form – needing only one parameterised graph (rather than 16 static graphs) — leads to the following, after only some trivial spider fusion:



Cutting lastly vertex #8 results in the following:



The graph at this stage contains 5 parameters (a, b, c, d, e) and 3 free nodes (legless spiders). As the parameters are Boolean, it necessarily follows that each of these free nodes can be replaced with a scalar factor of 0 or 2, such as follows:

$$(a + b + e)\pi = \begin{cases} 2 & \text{if } a \oplus b \oplus e = 0 \\ 0 & \text{if } a \oplus b \oplus e = 1 \end{cases} \quad (5.12)$$

Moreover, for any combination of parameter values that results in a scalar factor of 0 in one or more of these free nodes, one can ignore the entire corresponding graph (as the whole graph then becomes 0). In other words, one is only interested in the sets of parameter values which result in non-zero scalar factors for all free nodes. So, given equation 5.12, the leftmost free node must equal 2 and thus $a \oplus b \oplus e = 0$. Rearranging this for, say, a gives: $a = b \oplus e$. And now, every instance of a throughout the parameterised graph can be substituted out for $b \oplus e$, thus reducing the number of parameters from 5 to 4. Repeating this reasoning for the remaining two free nodes finds that $b = c \oplus e$ and $c = d \oplus e$, resulting in all parameters being reduced to some combination of d and e :

Consequently, while only needing to reason on one graph, the number of parameters has been reduced from the 5 attained via a suboptimal cut ordering to the most optimal 2. Indeed, this graph is now equivalent to what would have been attained if the more optimal cut ordering had been adopted (namely cutting vertex #8 first and then fusing the remaining 4 before cutting them as one). Having effectively corrected suboptimal cut ordering, one can then expand the parameterised graph out into its (in this case 4) distinct evaluated graphs and proceed with simplification.

Note, however, that the procedure outlined in section 5.2 prioritises higher tiers so that the cut ordering is already optimised and thus this parametric reasoning should not generally be necessary. Nevertheless, it might prove beneficial to the procedure in extreme cases, and indeed is applicable in other cutting techniques (such as the brute-force verifications of section 5.3.2).

5.2 Optimised Cutting Procedure

The solution this chapter presents is a procedure based on assigning weights to vertices, determined by how many T-like gates they are preventing from fusing to Clifford, and then propagating these weights through any neighbours which are then preventing weighted vertices from fusing, and so on up the tiers. Particular care is given to balance the weightings appropriately, especially in places where multiple ‘*spousal*’ cuts are required to facilitate a fusion of their children.

Note that one may label the weight of a vertex (i.e. spider) v , for a particular tier t , as w_v^t , such that, for instance, w_{12}^2 refers to the weight of vertex 12 with respect to tier 2. Given this, the procedure steps are as follows:

1. Partially simplify the circuit such that any instances of spider fusion are

applied (so no like-coloured spiders remain directly connected via a solid edge) and any π -phase spiders are pushed to one side or into CNOTs via the π -commutation and fusion rules. Then, let $t = 0$ and assign an initial weight of 0 to every spider (i.e. let $w_v^0 = 0 \forall v$).

2. For any pair of T-like spiders that are separated by k (that is, one or more) CNOTs, add $2/k$ to the weights of the opposite ends of each of those CNOTs. (For any given CNOT, take care not to count a particular T-spider more than once.) Now, for instance, any CNOT that is preventing 2 T-spiders from fusing to Clifford will have a corresponding weight of 2, and any CNOT preventing, collectively, 4 T-spiders from reducing to Cliffords will have a corresponding weighting of 4, etc. Similarly, if, for instance, 3 CNOTs are collectively blocking a single pair of T-spiders from fusing, then each will have a weighting of $2/3$.
3. Increment $t \leftarrow t + 1$. Then, similar to step 2, for any *weighted* vertex v of the previous tier (i.e. any v for which $w_v^{t-1} \geq 0$) that is separated from fusing with another weighted vertex of *any* lower tier (i.e. any v for which $w_v^u \geq 0$ for any $u < t$) by k (that is, one or more) CNOTs, add $\gamma(w_v^{t-1})/k$ to the weight of the opposite ends of each of those CNOTs. Here, the γ function normalises a given weighting to the range $[0, 1]$, such that (crudely speaking) 0 roughly means “very unlikely to be a worthwhile cut” and 1 “very likely to be a worthwhile cut”: $\gamma(w) := \min(\frac{w}{2}, 1)$.
4. Repeat step 3 until no new changes are made (that is, until no weightings, w_v^t , are found for any v , given the current t) or until a pre-defined tier limit is reached. At this point, one will be left with weightings for every vertex, for every tier ($w_v^t \forall v, t$). From this, one can trivially extract the relevant data, namely, for every vertex v , its *maximum* weight w_v^t (for any t) and,

respectively, the maximum t for which the vertex has a weight (i.e. largest t for which $w_v^t \geq 0$). One may label the maximum weight of a vertex, W_v , and its maximum tier for which it has a non-zero weight, T_v .

5. Among the vertices with the largest recorded T , namely T_{max} , select the one with the greatest max weight W_v (i.e. select vertex V s.t. $W_V \geq W_v \forall v$ s.t. $T_v = T_V = T_{max}$). (Note that for this step only, an additional weight of 1 may be added to any vertex of a T-like phase.) If this weight is below 2 (hence $\gamma(W_V) \leq 1$, implying the cut would not likely be worthwhile), then search instead among the vertices of the lower tier (i.e. for which $T_v = T_{max-1}$), until an appropriate vertex is found for which $W_V \geq 2$. This is the vertex which the heuristic has concluded is likely an optimal choice to cut. As such, cut this vertex (i.e. decompose it into two branches as per section 2.3.4).
6. Partially simplify each branch, without compromising the graph structure. Specifically, push any new π -phase spiders to one side, and/or into CNOTs, via repeated applications of the π -commutation rule (and fusion), and thereafter apply the fusion rule until no like-coloured spiders remain connected via a solid edge. Moreover, when fusing weighted spiders, update their combined weight accordingly as the sum of their respective max weights (i.e. in fusing vertex B into vertex A , the former is removed along with its recorded weightings and tier data and the latter is updated as $W_A \leftarrow W_A + W_B$). Similarly, the max tier of the newly fused vertex takes that of the larger of the two fused vertices (i.e. $T_A \leftarrow \max(T_A, T_B)$). Moreover, recalculate the weightings on any vertices whose children have been altered by this partial simplification.
7. Repeat steps 5 and 6 until no further cuts are made (or the number of T-

spiders ≤ 2). If any T-like spiders remain, then these may be decomposed via a typical T-decomposition (e.g. the BSS decomposition outlined in section 2.3.4).

Applying this procedure to a scalar Clifford+T ZX-diagram will result in a (heuristically) minimal number of stabiliser terms.

Python code, based on the PyZX package (Kissinger & van de Wetering, 2018, 2020a), that implements this procedure may be found at <https://github.com/mjsutcliffe99/ProcOptCut> (Sutcliffe, 2024b), and a step-by-step illustrative example of this procedure in action is shown therein.

5.3 Results

5.3.1 Circuit Generation

The procedure presented in this chapter, by design, works best on circuits that are highly structured. This raised an interesting problem when benchmarking, as testing on wholly random circuits would not properly showcase its effectiveness and, conversely, demonstrating only ideal example cases would not yield particularly informative results as such circuits could be made arbitrarily ideal, sending $\alpha \rightarrow 0$. Consequently, for the benchmarking experiments, random circuits were generated in such a way as to include some localised structural elements, so as to avoid trivial (unstructured) cases, while also not simply designing ‘best case’ circuits on which to experiment. As such, Toffoli gates were prominently utilised.

Specifically, the main benchmarking tests (figure 5.1) consider randomly generated circuits of the $\{CNOT, Z_\alpha, \text{Toffoli}\}$ gateset, where Z_α are Z-phase gates of phase $\alpha = \frac{n\pi}{4}$ with $n \in \{0, 1, 2, \dots, 7\}$. The number of such instances of these components was varied in order to vary the T-counts of the circuits. Mean-

5.3.2 Complexity and Efficiency

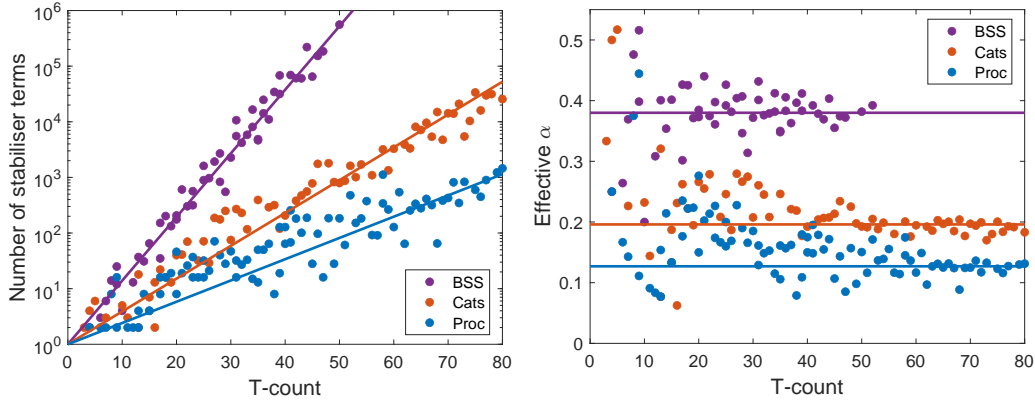
To test the effectiveness of the proposed method, experiments were run to explore how often - for circuits small enough to verify - it was able to find the *most* optimal set of vertex cuts, rather than simply *an* optimal set. Specifically, many random,

non-trivially structured (see section 5.3.1) ZX-diagrams of 16 internal Z-spiders or fewer were generated and fully decomposed using the procedure of section 5.2, with the number of stabiliser terms produced in each case measured. Given the small circuit sizes in this dataset, it was possible to test every possible combination of vertex cuts on the internal Z-spiders (applying the reasoning of section 5.1.4 to correct for any suboptimal cut ordering). Thus, it was possible to determine with certainty the most optimal set of vertex cuts (on Z-spiders) and correspondingly the number of stabiliser terms produced (or, by extension, the effective α) and compare this, in each case, to the result achieved by the procedural method. In this way, it was observed that this chapter’s method found the most optimal set of cuts possible (on Z-spiders) 71% of the time.

Even so, in every case in which the method failed to find the most optimal solution, it still invariably found a very good solution with an α_{eff} usually only marginally above that of the best. It is worth noting, however, that for larger circuits, too big to manually verify by brute force, it is very unlikely that this rate will be upheld. Nevertheless, this at least provides a broad indication of the method’s ability to find effective solutions. Furthermore, as the circuit scales are increased and scope for greater structure is enabled, it is expected that lower α_{eff} solutions will become more likely. As such, the analysis was extended to compare the results achieved by this method on larger circuits, versus the conventional approaches of (Kissinger & van de Wetering, 2022) and (Kissinger et al., 2022).

5.3.3 Experimental Measurements for Random Circuits

The procedural method presented in this chapter was benchmarked against the conventional direct decomposition approaches of (Kissinger & van de Wetering, 2022) and (Kissinger et al., 2022), for a dataset of randomly generated circuits of the form of section 5.3.1. In each case, the number of stabiliser terms to which



(a) Number of stabiliser terms versus T-count. (b) Effective α efficiency versus T-count.

Figure 5.1: (a) The number of stabiliser terms, n , produced after decomposing all t T-gates, and correspondingly (b) the effective overall decomposition efficiency α (given by $\alpha = \frac{1}{t} \log_2 n$) for numerous randomly generated Clifford+T circuits (see section 5.3.1) of various T-counts. In each case, the experimental results are shown for the conventional ‘BSS’ (Kissinger & van de Wetering, 2022) and ‘Cats’ (Kissinger et al., 2022) methods, as well as the procedurally optimised cutting (‘Proc’) approach presented in this chapter.

the circuit may be reduced was measured and, from this, the corresponding α_{eff} deduced. These results are shown in figure 5.1. These plots show the measured results from each individual circuit, via each of the three methods, as plotted against the initial (post Clifford simplification) T-counts of these circuits.

For very small T-counts, each method shows a lot of inconsistency in its effectiveness and generally produces worse α_{eff} measurements than at higher T-counts. This is unsurprising, as such circuits are likely too small to exhibit much structure and indeed small enough in T-count to rely more heavily on less efficient backup decompositions. For instance, when the T-count falls below 6, the BSS method relies upon a smaller, higher α , decomposition. Similar is true of the other methods. The especially effective results in this low T-count range, meanwhile, are presumably due to such small circuits being more likely to fully reduce after just one or

two applications of decomposition, as the simplicity of small circuits generally leads to easier simplification via the rewriting rules.

Regardless, circuits of very small T-count are trivially simulable and as such the results in this range are of little interest. Importantly, as the T-count is increased, each method, and particularly cats, becomes considerably more consistent in its effectiveness. Given this, it is sensible to consider the average α_{eff} that each method achieves for circuits of T-counts ≥ 40 , where their results become more stable. These averaged results are shown on the figures as lines, and may be quoted as:

$$\begin{aligned}\alpha_{\text{eff}}^{bss} &\approx 0.380 \\ \alpha_{\text{eff}}^{cats} &\approx 0.196 \\ \alpha_{\text{eff}}^{proc} &\approx 0.127\end{aligned}\tag{5.13}$$

It is already known (Kissinger et al., 2022) that the cats decomposition strategy supersedes the older BSS strategy, and this conclusion evidently remains true on these $\{CNOT, Z_\alpha, \text{Toffoli}\}$ circuits. The more interesting comparison then is that between the (presently state of the art) cats approach and the procedural cutting approach offered in this chapter.

The $\alpha_{\text{eff}}^{cats} \approx 0.196$ result measured here is very consistent with $\alpha_{\text{eff}}^{cats} \approx 0.205$ measured for Clifford+T+CCZ circuits in (Ahmad, 2024). This is sensible as the random circuits benchmarked in this chapter are indeed very similar to random Clifford+T+CCZ circuits, with Toffoli gates and CCZ gates being only marginally distinct when expressed in ZX-diagram form. Likewise, the low variance exhibited by the cats results here (beyond trivial T-counts) reinforces the observations of (Kissinger et al., 2022).

The most significant observation among these results is the promising performance of the new procedural method introduced in this chapter. Based on figure 5.1 and the result of $\alpha_{\text{eff}}^{\text{proc}} \approx 0.127$, this method appears to outperform the cats strategy quite reliably and significantly. To quantify more concretely, the cats strategy provides a 48% reduction in α_{eff} compared to the preceding BSS method, and in turn the new procedural approach offers a further 35% reduction from cats. Recalling that the runtime is proportional to the number of terms, which grows exponentially with α_{eff} , such improvements are very significant and of much practical interest. The new method would thus enable much larger T-count circuits to be simulable within computable timeframes.

There are a number of reasons which may help explain why an approach based solely on vertex cutting was able to outperform a strategy utilising more sophisticated decompositions. Firstly, the cats method is essentially a greedy algorithm with a very localised scope. At each step, it opts for the decomposition with the highest immediate α , without regard for how much or little simplification it is likely to facilitate. The procedural method, on the other hand, motivates its choice at each step very thoroughly, taking on board how much simplification (specifically T-count reduction) each possible move is liable to produce. This takes into account a much broader scope than the immediate vicinity of the cuts themselves.

Furthermore, as emphasised in section 5.1, the cutting decomposition's simplicity in practice works to its advantage. It is very effective at facilitating further simplification, particularly in the specific circumstances of which this method is designed to take advantage, and is applicable to any vertex in any ZX-diagram, giving much choice at each step.

Moreover, the procedure of section 5.2 is specifically designed for structured circuits, while strategies such as cats is more general and universal. It is perhaps to

be expected then that the procedural method proved to be more effective on highly structured circuits.

One final point to note is that the procedural method exhibits higher variance in its results as compared to the cats method. This is believed to be due to randomness of the circuit generation providing some variance in how structured they are. Even so, there is some indication to suggest the procedural method may reduce in variance as the T-count is further increased. Nevertheless, this greater variance also means there are many instances in which the procedural method is even more effective than suggested by the above quoted average, with $\alpha_{\text{eff}} < 0.1$ being reasonably common.

These experiments may be reproduced from the related Github repository (Sutcliffe, 2024b).

5.4 Conclusions

The main conclusion of this work is that, contrary to the assumption implicit in the literature, finding lower α decompositions is not the only, nor even necessarily the most effective, means of improving the efficiency of classical simulation via stabiliser decomposition. Instead, as demonstrated in this chapter, better motivating applications of existing decompositions shows arguably more promise.

By analysing, and taking advantage of, the structures inherent in any given quantum circuit, rather than applying decompositions arbitrarily, the method presented in this chapter is able to automatically optimise its decomposition strategy for the specific circuit. Consequently, despite its reliance on the apparently inefficient ‘cutting’ decomposition, this ensures a more optimal *overall* α_{eff} efficiency by predicting the extent of ZX-calculus simplification that each vertex cut may

facilitate (whether immediately or many steps ahead).

Applied to randomly generated circuits small enough to verify with brute force, this method found the *most* optimal vertex cutting pattern 71% of the time, with the remaining instances still finding highly optimal patterns. While it is presumed that on larger circuits it will become increasingly unlikely to find the *most* optimal patterns, the results suggest it continues to find highly optimal solutions. Indeed, in quantifying the effectiveness of the method on larger circuits, it is found to achieve an efficiency of $\alpha_{\text{eff}} \approx 0.127$, being a substantial improvement over the $\alpha_{\text{eff}} \approx 0.196$ achieved by (Kissinger et al., 2022) on the same dataset. This represents an exponential improvement to runtime for classically simulating such circuits.

While already very effective, there are many ways in which this method could be improved - many stemming from the rigid scope with which it applies and propagates weights. After all, T-gate fusion facilitated by cutting a CNOT is just one way in which the T-count of a circuit may be reduced. It would be worth considering also the potential of cuts to remove T-gates by pushing them into the scalar factor (e.g. via the state copy rule) or to partition small segments from the graph. The method ought also to consider simplification and weight propagation laterally, rather than solely through the lens of ‘tiers’. Indeed, the ‘partial simplification’ strategy used (to simplify while maintaining the structure) does not take into account some vital steps in the normal ‘full reduce’ (Kissinger & van de Wetering, 2018) function (namely pivoting - i.e. on CNOTs - and local complementation). It is suspected that this is largely the reason for not observing even higher success rates in verifying how often the method was able to find the best solutions on small circuits. Moreover, a more robust analysis could determine appropriate weightings *without* the need for the circuit to be expressed in a very rigid graph-like form, such that further simplification between steps could be enabled and, for

instance, Toffolis could be expressed as phase gadgets (so as to not be restricted due to the arbitrary choice of which way around to decompose each Toffoli's control qubits). And lastly, of course, one could consider cuts on X-spiders as well as just Z-spiders (this might be particularly relevant if there are many Hadamards involved, resulting in many X-spiders of T-like phase).

There are also a number of ways in which this concept, more broadly, could be improved, such as developing newer and better heuristics - perhaps even different heuristics for different types of circuit (e.g. dense circuits, or those with many Toffolis, etc.). Moreover, utilising additional decompositions is a promising avenue for improvement. In particular, the *cat* decompositions (Kissinger et al., 2022) possess some very helpful behaviours. For instance, decomposing cat states is very effective at producing more cat states. This likely helps explain the consistency and low variance of the method's effectiveness, as well as its ability to avoid relying significantly on its backup $|T\rangle^{\otimes 5}$ decomposition. As such, it is a promising candidate for the type of approach offered in this chapter.

Nevertheless, this work demonstrates very clearly how analysing the circuit structure and applying decompositions discriminately can offer vastly more efficient results than simply decomposing the T-spiders directly with a decomposition that has a better *immediate* efficiency.

6 | Dynamic T-Decomposition

Chapter 5 demonstrated how well motivated applications of weaker (lower α) T-decompositions can in fact lead to more efficient overall results following Clifford simplification via ZX-calculus rewriting. This challenged the broad assumption in the literature that the means to improving efficiency is by finding lower α decompositions.

The present chapter expands upon this by identifying a handful of circuit structures which occur frequently in common circuit classes (when expressed in reduced gadget form) and which may be decomposed efficiently with some rewriting and vertex cutting. This is ultimately expressed as a set of four new T-decompositions which, in one sense, are more specific than the likes of the cats decompositions of equation 2.61 in that the structures involved are less generic, but which, in another sense, are more dynamic in that they are arbitrarily scalable.

Due to their scalability, the α efficiencies of these decompositions depend upon the number of spiders involved, though this is easily measured and can be used to heuristically determine which decomposition is likely to be most effective at any given moment. In short, this chapter outlines a general heuristic approach to efficient Clifford+T circuit reduction.

Lastly, these new decompositions are benchmarked against the method of (Kissinger et al., 2022) for four different circuit classes, with results showing significant improvements to runtime in three of the four cases.

This chapter covers the work presented in (Ahmad & Sutcliffe, 2024), which in turn expands upon a master's project (Ahmad, 2024) I proposed and supervised during my DPhil studies.

6.1 Deriving Dynamic Decompositions

Scalar Clifford+T ZX-diagrams, simplified to reduced gadget form, tend to exhibit common structures and patterns. Consequently, there is much justification for designing stabiliser decompositions which take advantage of this fact. The *cat* decompositions (Kissinger et al., 2022), for example, utilise $|\text{cat}_n\rangle$ states, which are characteristic of reduced gadget graphs, and achieve impressive α efficiencies. However, with $|\text{cat}_n\rangle$ states being rather generic and minimalistic structures, their corresponding decompositions fail to take advantage of more specific and niche patterns which, paired with very simple decompositions, are liable to undergo more drastic simplification.

By manual analysis, four such decompositions were derived, each based upon one instance of vertex cutting (figure 2.68) and, in the latter two cases, some initial rewriting. These decompositions, together with their derivations, are shown in lemmas 30 to 33. For convenience, they may be divided into two categories:

- **Singled** decompositions, being those of lemmas 30 and 31, which are each derived via one instance of vertex cutting, and
- **Doubled** decompositions, being those of lemmas 32 and 33, which are each derived via an initial instance of pivoting, followed by a vertex cut.

The former describe structures which are decomposed directly via an instance of vertex cutting, while the latter require an initial simplification step (namely pivoting) before undergoing a vertex cut.

Lemma 30 (Lone phase decomposition). *The following decomposition:*

$$\begin{array}{c} x_1 \quad x_n \\ \circ \quad \circ \\ \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \\ \circ \quad \circ \\ \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \end{array} \approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \left(1 + e^{i\pi(\frac{1}{4}+a)}\right)^n
 \begin{array}{c} a\pi \quad a\pi \\ \vdots \quad \vdots \end{array} \quad (6.1)$$

is valid and achieves:

$$\alpha = \frac{1}{n+1} \quad (6.2)$$

Proof.

$$\begin{array}{c} x_1 \quad x_n \\ \circ \quad \circ \\ \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \\ \circ \quad \circ \\ \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \end{array} \approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \begin{array}{c} x_1 \quad x_n \\ \circ \quad \circ \\ \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \\ a\pi \quad a\pi \\ \vdots \quad \vdots \end{array} \quad (6.3)$$

$$\approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \begin{array}{c} x_1 \quad x_n \\ \pi/4 + a\pi \quad \pi/4 + a\pi \\ \vdots \quad \vdots \\ a\pi \quad a\pi \\ \vdots \quad \vdots \end{array}$$

$$\approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \left(1 + e^{i\pi(\frac{1}{4}+a)}\right)^n \begin{array}{c} a\pi \quad a\pi \\ \vdots \quad \vdots \end{array}$$

□

Lemma 31 (Multi- $|\text{cat}_3\rangle$ decomposition (Codsì, 2022)). *The following decomposition:*

$$\begin{array}{c} x_1 \quad x_2 \quad x_n \\ \circ \quad \circ \quad \circ \\ \pi/4 \quad \pi/4 \quad \pi/4 \\ \vdots \quad \vdots \quad \vdots \\ v \quad y_1 \quad y_2 \quad \dots \quad y_n \\ \circ \quad \circ \quad \circ \quad \dots \quad \circ \\ \pi/4 \quad \pi/4 \quad \pi/4 \quad \dots \quad \pi/4 \\ \vdots \quad \vdots \quad \vdots \quad \dots \quad \vdots \end{array} \approx \sum_{a \in \{0,1\}} e^{ia(n+1)\frac{\pi}{4}} \begin{array}{c} y_1 \quad y_2 \quad \dots \quad y_n \\ (1-a)\frac{\pi}{2} \quad (1-a)\frac{\pi}{2} \quad \dots \quad (1-a)\frac{\pi}{2} \\ \vdots \quad \vdots \quad \dots \quad \vdots \end{array} \quad (6.4)$$

$$\alpha = \frac{1}{2n+1} \quad (6.5)$$

Proof.

☐

Lemma 32. *The following decomposition:*

$$\begin{array}{c}
\begin{array}{c} \cdots \\ \textcircled{\frac{\pi}{4}} \\ x_1 \end{array} \quad \begin{array}{c} y_1 \\ \textcircled{\frac{\pi}{4}} \end{array} \quad \begin{array}{c} y_n \\ \textcircled{\frac{\pi}{4}} \end{array} \quad \begin{array}{c} \cdots \\ \textcircled{\frac{\pi}{4}} \\ x_n \end{array} \\
\vdots \quad \vdots \quad \vdots \quad \vdots \\
\begin{array}{c} z_1 \\ \textcircled{\frac{\pi}{4}} \end{array} \quad \cdots \quad \begin{array}{c} z_2 \\ \textcircled{\frac{\pi}{4}} \end{array} \quad \cdots
\end{array}
\approx \sum_{a \in \{0,1\}} e^{ia(n+1)\frac{\pi}{4}}
\begin{array}{c}
\cdots \quad \cdots \\
\textcircled{(1-a)\frac{\pi}{2}} \quad \cdots \quad \textcircled{(1-a)\frac{\pi}{2}} \\
x_1 \quad \quad \quad x_n \\
\vdots \quad \vdots \quad \vdots \\
\textcircled{(1-a)\frac{\pi}{2}} \\
\vdots \quad \vdots \quad \vdots
\end{array}
\quad (6.7)$$

is valid and achieves:

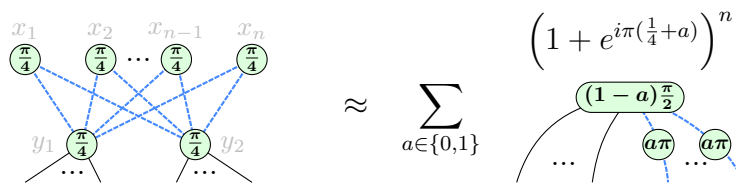
$$\alpha = \frac{1}{2n+2} \tag{6.8}$$

Proof.

$$\begin{aligned}
 & \begin{array}{c} \text{Diagram 1: A network with nodes } x_1, \dots, x_n \text{ (top) and } z_1, \dots, z_2 \text{ (bottom). Each node is a green circle with } \frac{\pi}{4}. \text{ Blue dashed lines connect } x_1 \text{ to } z_1 \text{ and } x_n \text{ to } z_2. \text{ Intermediate nodes } y_1, \dots, y_n \text{ are also green circles with } \frac{\pi}{4}. \end{array} \\
 & \approx \begin{array}{c} \text{Diagram 2: Similar to Diagram 1, but with a central node } v \text{ (green circle with } \frac{\pi}{4}) \text{ connected to } y_1, \dots, y_n \text{ and } z_1, \dots, z_2. \end{array} \\
 & \approx \sum_{a \in \{0,1\}} \begin{array}{c} \text{Diagram 3: Similar to Diagram 2, but with nodes } a\pi \text{ (green circles) instead of } \frac{\pi}{4} \text{ for } y_1, \dots, y_n \text{ and } z_1, \dots, z_2. \end{array} \quad (6.9) \\
 & \approx \sum_{a \in \{0,1\}} \begin{array}{c} \text{Diagram 4: Similar to Diagram 3, but with nodes } a\pi \text{ (green circles) instead of } \frac{\pi}{4} \text{ for } x_1, \dots, x_n \text{ and } z_1, \dots, z_2. \end{array} \\
 & \approx \sum_{a \in \{0,1\}} e^{ia(n+1)\frac{\pi}{4}} \begin{array}{c} \text{Diagram 5: Similar to Diagram 4, but with nodes } \frac{\pi}{4} \text{ (green circles) instead of } a\pi \text{ for } x_1, \dots, x_n \text{ and } z_1, \dots, z_2. \end{array} \\
 & \approx \sum_{a \in \{0,1\}} e^{ia(n+1)\frac{\pi}{4}} \begin{array}{c} \text{Diagram 6: Similar to Diagram 5, but with nodes } (1-a)\frac{\pi}{2} \text{ (green circles) instead of } \frac{\pi}{4} \text{ for } x_1, \dots, x_n \text{ and } z_1, \dots, z_2. \end{array}
 \end{aligned}$$

□

Lemma 33. *The following decomposition:*


$$\approx \sum_{a \in \{0,1\}} \left(1 + e^{i\pi(\frac{1}{4}+a)}\right)^n$$

(6.10)

is valid and achieves:

$$\alpha = \frac{1}{n+2}$$

(6.11)

Proof.

$$\begin{aligned}
 & \begin{array}{c} x_1 \quad x_2 \quad x_{n-1} \quad x_n \\ \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad \dots \quad \left(\frac{\pi}{4} \right) \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \end{array} \approx \begin{array}{c} x_1 \quad x_2 \quad x_{n-1} \quad x_n \\ \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad \dots \quad \left(\frac{\pi}{4} \right) \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ v \\ \vdots \\ y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \end{array} \\
 & \approx \sum_{a \in \{0,1\}} \begin{array}{c} x_1 \quad x_2 \quad x_{n-1} \quad x_n \\ \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad \dots \quad \left(\frac{\pi}{4} \right) \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ a\pi \quad a\pi \quad \dots \quad a\pi \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \end{array} \\
 & \approx \sum_{a \in \{0,1\}} \begin{array}{c} x_1 \quad x_2 \quad x_{n-1} \quad x_n \\ \left(\frac{\pi}{4} + a\pi \right) \quad \left(\frac{\pi}{4} + a\pi \right) \quad \dots \quad \left(\frac{\pi}{4} + a\pi \right) \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \end{array} \\
 & \approx \sum_{a \in \{0,1\}} \begin{array}{c} \left(1 + e^{i\pi(\frac{1}{4}+a)} \right)^n \\ y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \end{array} \\
 & \approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \left(1 + e^{i\pi(\frac{1}{4}+a)} \right)^n \begin{array}{c} y_1 \quad \left(\frac{\pi}{4} \right) \quad \left(\frac{\pi}{4} - a\frac{\pi}{2} \right) \quad y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ a\pi \quad a\pi \end{array} \\
 & \approx \sum_{a \in \{0,1\}} e^{ia\frac{\pi}{4}} \left(1 + e^{i\pi(\frac{1}{4}+a)} \right)^n \begin{array}{c} (1-a)\frac{\pi}{2} \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ a\pi \quad a\pi \end{array}
 \end{aligned}
 \tag{6.12}$$

□

These decompositions apply to patterns which, while less ubiquitous than $|\text{cat}_n\rangle$ states, are relatively common among ZX-diagrams in reduced gadget form. They are patterns for which a single vertex cut enables significant simplification via the rewriting rules, and, importantly, they are dynamic, in that they are each arbitrarily scalable.

Note that, as with applying the $|\text{cat}_n\rangle$ decompositions, the following lemma holds true here:

Lemma 34. *The decompositions introduced in lemmas 30 to 33 each remain valid (albeit with minor changes to scalars and/or spider phases) with any number of T -spiders ($\textcircled{\frac{\pi}{4}}$) replaced with T -like spiders ($\textcircled{(2n+1)\frac{\pi}{4}}$).*

Algorithm 6 summarises the decision process of selecting which decomposition to apply at each step.

6.2 Results

Using algorithm 6, these new dynamic decompositions were benchmarked against the existing method of Kissinger et al. (Kissinger et al., 2022) for strongly simulating randomly generated circuits of the following four distinct classes:

- Controlled-CZ (CCZ) circuits
- Modified hidden shift (MHS) circuits
- Instantaneous Quantum Polynomial (IQP) circuits
- Pauli exponentials circuits

These circuit classes are detailed in section 2.2.3.

Algorithm 6 Algorithm for determining which decomposition to apply

```

1: Input: A ZX-diagram,  $G$ , in reduced gadget form
2:
3: Function:  $\alpha(d)$  returns the  $\alpha$  value corresponding to decomposition  $d$ 
4: Let  $\alpha_{\text{best}} = \alpha(|\mathbf{T}\rangle^{\otimes 5}) = \log_2(3)/4$ 
5:
6: if there exists an instance of  $|\text{cat}_n\rangle$  in  $G$  for any  $3 \leq n \leq 6$  then
7:   Let  $|\text{cat}_{\text{best}}\rangle$  be the lowest- $\alpha$   $|\text{cat}_n\rangle$  that exists within  $G$ 
8:   if  $\alpha(|\text{cat}_{\text{best}}\rangle) < \alpha_{\text{best}}$  then
9:     Let  $\alpha_{\text{best}} = \alpha(|\text{cat}_{\text{best}}\rangle)$ 
10:   end if
11: end if
12:
13: for each decomposition  $D_n$  among those of lemmas 30 to 33 do
14:   if there exists an instance of  $D_n$  in  $G$  for any  $n \geq 1$  then
15:     Let  $N$  be the largest  $n$  for which an instance of  $D_n$  exists within  $G$ 
16:     if  $\alpha(D_N) < \alpha_{\text{best}}$  then
17:       Let  $\alpha_{\text{best}} = \alpha(D_N)$ 
18:     end if
19:   end if
20: end for
21:
22: Output: The decomposition instance corresponding to  $\alpha_{\text{best}}$ 

```

The results of these experiments are displayed in figures 6.1 to 6.4, showing how both the decomposition efficiency and the measured runtime vary with the scale of the circuit. In each case, the results are shown for:

- the Kissinger et al. (Kissinger et al., 2022) decompositions only,
- the above plus the *singled* dynamic decompositions introduced in this chapter, and
- the above plus also the *doubled* dynamic decompositions introduced in this chapter.

Note that, consistent with the literature (Codsí & van de Wetering, 2022; Ahmad, 2024), the effective decomposition efficiency for IQP circuits is measured with β_{eff} rather than the usual α_{eff} . This is a more appropriate metric¹ for benchmarking such circuits as it neglects the trivially handled T-gates and provides a neat measure against the qubit count, which such circuits notably scale against. This metric is defined as follows:

Definition 32 (β_{eff} efficiency). *Given a ZX-diagram in reduced gadget form, with t' T-like spiders which are not part of phase gadgets and $\tilde{O}((t')^k)$ phase gadgets, for $k \in \mathbb{R}_{>1}$, the effective β efficiency is defined:*

$$\beta_{\text{eff}} := \frac{N}{t'} \quad (6.13)$$

where N is the number of stabiliser terms to which the circuit is decomposed (Ahmad, 2024).

The first observation one might make is that the new method appears to never perform worse than that of Kissinger et al. (Kissinger et al., 2022) with regards to its effective α , but sometimes (though very seldom and marginally) does so

¹See (Ahmad, 2024) for a more explicit justification of this metric.

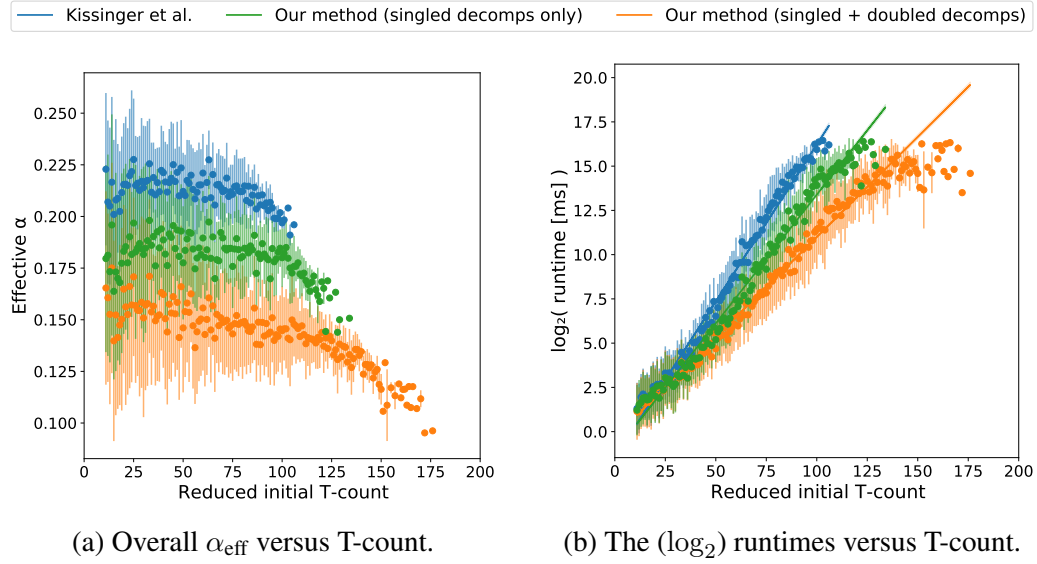


Figure 6.1: The measured results for classically simulating **CCZ** circuits, versus the method of Kissinger et al. (Kissinger et al., 2022)

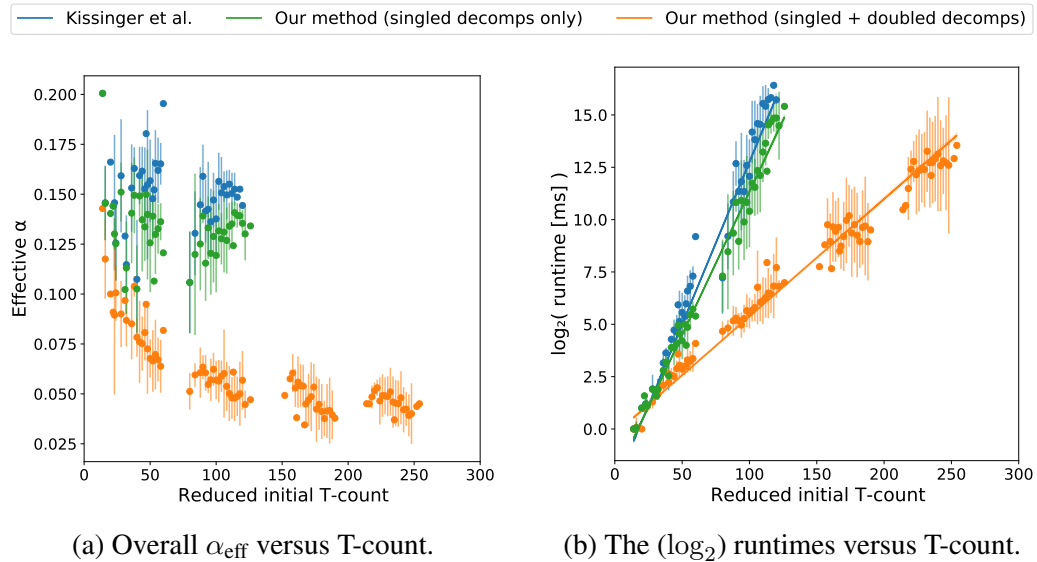


Figure 6.2: The measured results for classically simulating **modified hidden shift** circuits, versus the method of Kissinger et al. (Kissinger et al., 2022)

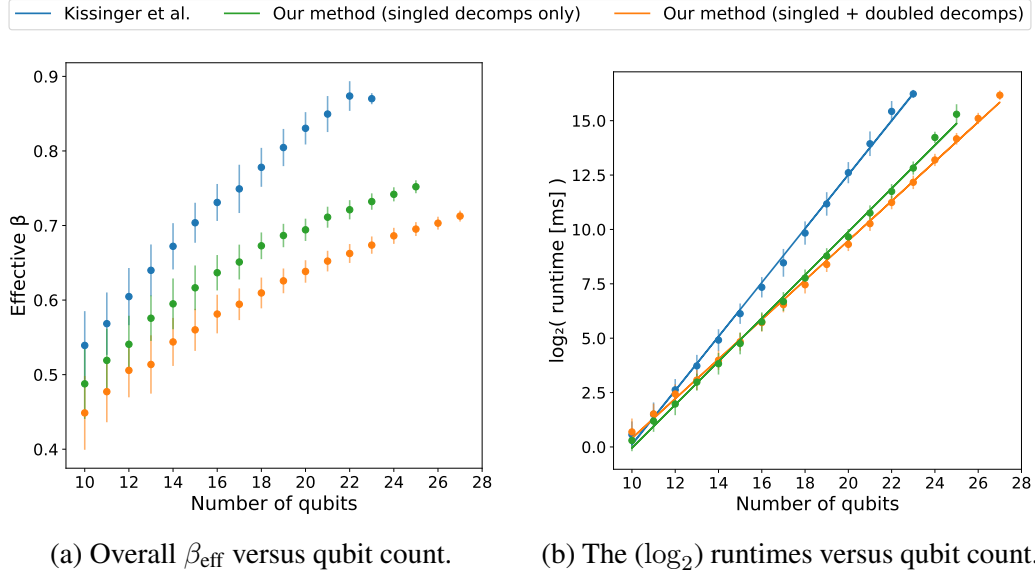


Figure 6.3: The measured results for classically simulating **IQP** circuits, versus the method of Kissinger et al. (Kissinger et al., 2022)

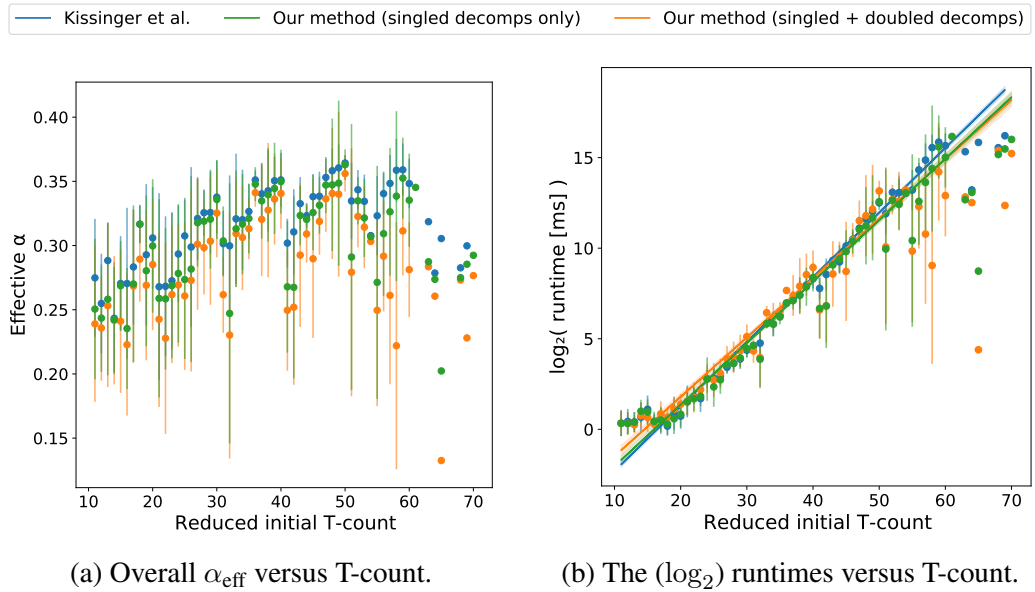


Figure 6.4: The measured results for classically simulating **random Pauli** circuits, versus the method of Kissinger et al. (Kissinger et al., 2022)

with regards to real runtime. This is explained by the fact that, as per algorithm 6, the new method theoretically supersedes the old in that it only applies the new dynamic decompositions if and when they are expected (as measured by α) to outperform the decompositions introduced in (Kissinger et al., 2022). Despite this, there remain two possible ways in which the new method can offer worse runtime results than the old:

- The α_{eff} of the new method is marginally lower than that of the old method, but the additional computational overhead in finding applications of the new decompositions makes up more than the difference in runtime.
- At a given step, one of the new decompositions offers the best α and is hence chosen, but a $|\text{cat}_n\rangle$ (or $|\text{T}\rangle^{\otimes 5}$) decomposition, despite its greater α , happens to lead to a better α_{eff} (for the current step) by enabling greater simplification. While theoretically possible, this seems to occur with exceptional rarity, if at all, and, amongst the experimental results presented here, was never sufficient to worsen the *overall* α_{eff} .

Fortunately, neither of these issues occur in practice to any degree of significance, except in cases of trivially small circuits.

It should be noted that both the trailing off visible at high T-counts in figure 6.1 and the gaps present in figure 6.2 are artifacts of the parameters used for the random circuit generation. Specifically, as both depth d and qubit count q were modified to vary the T-counts of the generated circuits, the former artifact is a consequence of circuits with particularly low $\frac{q}{d}$ ratios leading to uncharacteristically trivial ZX-diagrams after simplification. Meanwhile, the gaps in figure 6.2 are due to the number of Fredkin gates (from which the T-counts arise) being incremented in steps of 10, rather than a more granular increment. These artifacts should be overlooked and emphasis should be placed instead on the stable portions of these

figures.

Overall, these results show that the introduction of the four new dynamic decompositions invariably leads to significant reductions in runtime for strongly simulating three of the four circuit classes. Furthermore, the trends predict that as larger scale circuits are considered, these improvements are even more significant, and exponentially so. This is as would be expected from reducing α_{eff} (or β_{eff}), and shows that the potential added complexity of searching for applications of these dynamic decompositions proves not to be an obstacle.

Interestingly, the improvements are not uniform among the three circuit classes to which the new method was successful. Instead, one observes that the individual effectiveness of each dynamic decompositions varies noticeably with the circuit class. In particular, the singled dynamic decompositions appear to offer a significant improvement to the runtime of simulating CCZ circuits, with the inclusion of the doubled decompositions providing a comparable additional improvement. For modified hidden shift circuits, meanwhile, the inclusion of the singled decompositions results in only a small difference, though incorporating also their doubled counterparts leads to a substantial improvement. The converse appears to be true of IQP circuits, where a notable enhancement is seen by the singled decompositions, with the doubled cases making only a marginal difference. Lastly, neither set of dynamic decompositions seem to make a significant difference versus the existing method when applied to fully random circuits generated via Pauli exponentials. These observations are qualitatively summarised in table 6.1.

One may make sense of these results by closely considering the structures of these different circuit classes (see section 2.2.3) and how they relate to the structures among these dynamic decompositions. For instance, CCZ gates contain structures comparable to those of which the *doubled* decompositions take advan-

Circuit Class	Singled	Doubled
CCZ	✓	✓
MHS	—	✓
IQP	✓	—
Pauli	✗	✗

Table 6.1: A qualitative analysis of the dynamic decomposition sets, highlighting significant improvements (✓), marginal improvements (—), and negligible/no improvements (✗) in runtime rates for the different circuit classes.

tage, justifying the effectiveness of these decompositions on both CCZ and MHS circuits. However, without the additional Hadamard and CNOT gates (compare equations 2.40 and 2.43), CCZ circuits are more susceptible to simplification via the rewriting rules, producing reduced gadget diagrams with a greater ubiquity of such structures seen in the *singled* decompositions. This, it is suspected, explains why these decompositions are effective on CCZ circuits but less so when applied to MHS circuits.

Meanwhile, from equation 2.42, it is evident that IQP circuits are essentially just composed sets of multi- $|\text{cat}_3\rangle$ states. As such, it is unsurprising that the multi- $|\text{cat}_3\rangle$ decomposition (categorised as *singled*) is highly effective on such circuits. Lastly, Pauli circuits are the most random and unstructured of the classes circuits, which explains why these structure-specific dynamic decompositions are largely ineffective. Fortunately, being essentially fully random, these circuits are arguably the least realistic among the four classes under consideration.

6.3 Conclusions

This work identified a number of common circuit structures and patterns to which well-placed vertex cuts lead to significant reductions in T-count after ZX-calculus

simplification is applied along each branch. These patterns are formalised as dynamically structured T-decompositions with efficiency α calculated as a function of the scale of the pattern. Due to the dynamic nature of these decompositions, as compared to most alternatives in the literature (Kissinger & van de Wetering, 2022; Kissinger et al., 2022; Koch et al., 2023; Laakkonen, 2022), they are typically highly applicable and are often able to achieve very low α efficiencies.

Benchmarked against the state of the art decomposition strategy of (Kissinger et al., 2022), these dynamic decompositions are shown to be extremely effective on three of the four circuit classes considered. Moreover, the effectiveness of the individual decompositions is not uniform across these circuit classes, and this observation is analysed and explained through close inspection of the circuit structures involved in each case. Overall, significantly lower α_{eff} are achieved with these new decompositions, enabling classical simulation of CCZ, MHS, and IQP circuits of much greater T-count than would be achievable with the existing non-dynamic decompositions.

The results shown in this chapter relate to the specific dynamic decompositions introduced in lemmas 30 to 33. However, a more general conclusion from this chapter is the demonstration of the potential of such *dynamic* decompositions more broadly. Along these lines, and with further analysis of common circuit structures, a greater number of such decompositions may be discovered. Additionally, as this chapter demonstrates, such decompositions can be found through manual analysis and extending known decompositions (such as the cutting decomposition) to special cases, rather than relying on such catch-all techniques as simulated annealing.

7 | Conclusions and Future Directions

Classical simulation of quantum circuits is a vital tool for understanding the quantum advantage, verifying quantum software and hardware, and optimising quantum algorithms. However, it is a notoriously inefficient problem to solve, with resource requirements that grow exponentially with one or more metrics of the given quantum circuits. Various methods of classical simulation have been employed in recent years, including notably those of tensor contraction and stabiliser decomposition. To the latter method especially, the graphical language of ZX-calculus has been thoroughly applied, with a growing body of literature continuing to demonstrate newer, more efficient stabiliser decompositions. This research has steadily improved the exponential growth rate with which the computational complexity scales with the number of costly ‘*T-gates*’ in the circuit.

Building off of this existing literature, the work presented throughout this thesis introduces new techniques and methods by which quantum circuits may be classically simulated, with the aid of ZX-calculus. This includes:

- a parameterisation of ZX-calculus which allows many similar ZX-diagrams to be reduced as one and efficiently evaluated with the use of a GPU,
- a hybrid method, optimally combining stabiliser decomposition and tensor contraction approaches,
- a procedure for heuristically motivating decomposition choices, predicting the likely simplification facilitated in each case, to optimise for the *overall* efficiency, α_{eff} , and
- a set of new ‘*dynamic*’ stabiliser decompositions, derived from common special cases of the cutting decomposition.

These new methods are benchmarked against the comparable alternative approaches, particular via stabiliser decomposition, and in each case an appropriate analysis of the types of circuits upon which these methods excel is provided. For each of the above methods, the results indicate substantial improvements to the runtime for classical simulation of particular types of quantum circuits.

The parameterisation work of chapter 3 is widely applicable to a number of tasks within classical simulation and beyond, and is largely agnostic of the circuit metrics. It may be used in conjunction with the other techniques introduced and, even with a modest commercial GPU, offers linear runtime improvements near a factor 100.

Meanwhile, the hybrid method introduced in chapter 4 theoretically always outperforms (or at least matches) both stabiliser decomposition and tensor contraction, as both may be seen as special cases of this new method. In fact, in practice, this method is extremely effective — outperforming both naïve methods by many orders of magnitude — on circuits consisting of scattered cliques which are internally dense.

A broad conclusion of 5 is that deciding when and where to apply decompositions can be more important than deciding *which* decompositions to use. The weighted heuristic method presented in this chapter utilises only a very simple and trivial decomposition, though manages to improve upon the overall decomposition efficiency, reducing the runtime scaling factor against T-count t from $O(2^{0.196t})$ to $O(2^{0.127t})$.

The final content chapter (chapter 6) considers addition heuristics for deciding when and where to apply the cutting decomposition, expressed as new *dynamic* decompositions, which likewise improve upon the runtime complexity, with notable variance across circuit classes. On almost all circuit classes considered, this

resulted in drastic improvements to the overall efficiency.

Ultimately, the methods presented in this thesis seek to reduce the computational cost of simulating quantum circuits with classical hardware. This means reducing the runtime for simulating given circuits and improving the rate at which this computational cost scales with the size and complexity of the circuit. Importantly, this raises the limits of what is classically simulable, with ever larger and more complex circuits being rendered simulable within reasonable timeframes. This in turn has significant implications for understanding the quantum advantage and for verifying and optimising quantum software and hardware — which is vital in the NISQ era of today.

To this end, these methods prove very effective, with particular benefits and limitations of each. Some are more favourable for certain circuit classes, while others depend more on the shape of the circuit or structures therein. To some extent, these methods may be used to complement one another and may be used in conjunction with existing approaches, while others supersede the known alternatives under certain conditions. Moreover, in each case, the ideas demonstrated may be further expanded and built upon in future research, as has already been seen in some recent works (Ahmad, 2024; Koziell-Pipe et al., 2024; Vollmeier, 2025).

Overall, the research presented in this thesis expands the catalogue of tools available for classical simulation of quantum circuits, with proven and effective methods of practical utility. In addition to this, on a more theoretical note, these works highlight and demonstrate ideas and concepts which may shed light on how further developments and optimisations of classical simulation may follow.

No spiders were harmed in the making of this thesis.

Bibliography

- Aaronson, S. (2013). *Quantum computing since democritus*. Cambridge University Press.
- Aaronson, S., & Gottesman, D. (2004). Improved simulation of stabilizer circuits. *Physical Review A — Atomic, Molecular, and Optical Physics*, 70(5), 052328.
- Abdelfattah, A., Baboulin, M., Dobrev, V., Dongarra, J., Earl, C., Falcou, J., ... others (2016). High-performance tensor contractions for gpus. *Procedia Computer Science*, 80, 108–118.
- Acharya, R., Aghababaie-Beni, L., Aleiner, I., Andersen, T. I., Ansmann, M., Arute, F., ... others (2024). Quantum error correction below the surface code threshold. *arXiv preprint arXiv:2408.13687*.
- Aharonov, D. (2003). A simple proof that toffoli and hadamard are quantum universal. *arXiv preprint quant-ph/0301040*.
- Ahmad, W. A. (2024). *Efficient heuristics for classical simulation of quantum circuits using zx-calculus* (Master's thesis, University of Oxford). Retrieved from <https://www.cs.ox.ac.uk/people/aleks.kissinger/theses/ahmad-thesis.pdf>
- Ahmad, W. A., & Sutcliffe, M. (2024). Dynamic t-decomposition for classical simulation of quantum circuits. *arXiv preprint arXiv:2412.17182*.
- Alerstam, E., Svensson, T., & Andersson-Engels, S. (2008). Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of biomedical optics*, 13(6), 060504–060504.
- Althoby, H. Y., Biha, M. D., & Sesboüé, A. (2020). Exact and heuristic methods for the vertex separator problem. *Computers & Industrial Engineering*, 139, 106135.

- Backens, M. (2015). Making the stabilizer zx-calculus complete for scalars. *arXiv preprint arXiv:1507.03854*.
- Backens, M. (2016). Completeness and the zx-calculus. *arXiv preprint arXiv:1602.08954*.
- Barak, B., & Marwaha, K. (2021). Classical algorithms and quantum limitations for maximum cut on high-girth graphs. *arXiv preprint arXiv:2106.05900*.
- Berent, L., Burgholzer, L., & Wille, R. (2022). Towards a sat encoding for quantum circuits: A journey from classical circuits to clifford circuits and beyond. *arXiv preprint arXiv:2203.00698*.
- Bertsimas, D., & Tsitsiklis, J. (1993). Simulated annealing. *Statistical science*, 8(1), 10–15.
- Bharti, K., Cervera-Lierta, A., Kyaw, T. H., Haug, T., Alperin-Lea, S., Anand, A., ... Menke, T. (2022). Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1), 015004.
- Blekos, K., Brand, D., Ceschini, A., Chou, C.-H., Li, R.-H., Pandya, K., & Summer, A. (2024). A review on quantum approximate optimization algorithm and its variants. *Physics Reports*, 1068, 1–66.
- Boneh, D., et al. (1999). Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2), 203–213.
- Bravyi, S., Browne, D., Calpin, P., Campbell, E., Gosset, D., & Howard, M. (2019). Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, 3, 181.
- Bravyi, S., & Gosset, D. (2016). Improved classical simulation of quantum circuits dominated by clifford gates. *Physical review letters*, 116(25), 250501.
- Bravyi, S., Gosset, D., & Liu, Y. (2022). How to simulate quantum measurement without computing marginals. *Physical Review Letters*, 128(22), 220503.
- Bravyi, S., Smith, G., & Smolin, J. A. (2016). Trading classical and quantum

- computational resources. *Physical Review X*, 6(2), 021043.
- Bremner, M. J., Jozsa, R., & Shepherd, D. J. (2011). Classical simulation of commuting quantum computations implies collapse of the polynomial hierarchy. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 467(2126), 459–472.
- Bremner, M. J., Montanaro, A., & Shepherd, D. J. (2017). Achieving quantum supremacy with sparse and noisy commuting quantum computations. *Quantum*, 1, 8.
- Brennan, J., Allalen, M., Brayford, D., Hanley, K., Iapichino, L., O’Riordan, L. J., ... Moran, N. (2021). Tensor network circuit simulation at exascale. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)* (pp. 20–26).
- Bretto, A. (2013). *Hypergraph theory: An introduction*. Springer International Publishing. Retrieved from <https://books.google.co.uk/books?id=lb5DAAAQBAJ>
- Breuer, H.-P., & Petruccione, F. (2002). *The theory of open quantum systems*. OUP Oxford.
- Bruzewicz, C. D., Chiaverini, J., McConnell, R., & Sage, J. M. (2019). Trapped-ion quantum computing: Progress and challenges. *Applied physics reviews*, 6(2).
- Bui, T. N., & Jones, C. (1992). Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3), 153–159. Retrieved from <https://www.sciencedirect.com/science/article/pii/002001909290140Q> doi: [https://doi.org/10.1016/0020-0190\(92\)90140-Q](https://doi.org/10.1016/0020-0190(92)90140-Q)
- Callison, A., & Chancellor, N. (2022). Hybrid quantum-classical algorithms in the noisy intermediate-scale quantum era and beyond. *Physical Review A*,

106(1), 010101.

Carette, T., Horsman, D., & Perdrix, S. (2019). Szx-calculus: Scalable graphical quantum reasoning. *arXiv preprint arXiv:1905.00041*.

Chancellor, N., Kissinger, A., Zohren, S., Roffe, J., & Horsman, D. (2023). Graphical structures for design and verification of quantum error correction. *Quantum science and technology*, 8(4), 045028.

Chundury, S., Li, J., Suh, I.-S., & Mueller, F. (2024). Diaq: Efficient state-vector quantum simulation. *arXiv preprint arXiv:2405.01250*.

Codsi, J. (2022). *Cutting-edge graphical stabiliser decompositions for classical simulation of quantum circuits* [Master's thesis]. Retrieved from <https://www.cs.ox.ac.uk/people/aleks.kissinger/theses/codsi-thesis.pdf>

Codsi, J., & van de Wetering, J. (2022). Classically simulating quantum supremacy iqp circuits through a random graph approach. *arXiv preprint arXiv:2212.08609*.

Coecke, B., de Felice, G., Meichanetzidis, K., & Toumi, A. (2020). Foundations for near-term quantum natural language processing. *arXiv preprint arXiv:2012.03755*.

Coecke, B., de Felice, G., Meichanetzidis, K., Toumi, A., Gogioso, S., & Chiappori, N. (2020). Quantum natural language processing. *url: http://www.cs.ox.ac.uk/people/bob.coecke/QNLP-ACT.pdf*.

Coecke, B., & Duncan, R. (2008). Interacting quantum observables. In *International colloquium on automata, languages, and programming* (pp. 298–310).

Coecke, B., & Duncan, R. (2011). Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4), 043016.

Coecke, B., & Kissinger, A. (2018). Picturing quantum processes: A first course

- on quantum theory and diagrammatic reasoning. In *Diagrammatic representation and inference: 10th international conference, diagrams 2018, edinburgh, uk, june 18-22, 2018, proceedings 10* (pp. 28–31).
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
- Cowlessur, H., Thapa, C., Alpcan, T., & Camtepe, S. (2024). A hybrid quantum neural network for split learning. *arXiv preprint arXiv:2409.16593*.
- Cowtan, A., Dilkes, S., Duncan, R., Simmons, W., & Sivarajah, S. (2019). Phase gadget synthesis for shallow circuits. *arXiv preprint arXiv:1906.01734*.
- Crane, K., Llamas, I., & Tariq, S. (2007). Real-time simulation and rendering of 3d fluids. *GPU gems*, 3(1).
- Criminisi, A., Sharp, T., Rother, C., & Pérez, P. (2010). Geodesic image and video editing. *ACM Trans. Graph.*, 29(5), 134–1.
- De Beaudrap, N., Bian, X., & Wang, Q. (2019). Techniques to reduce $\pi/4$ -parity-phase circuits, motivated by the zx calculus. *arXiv preprint arXiv:1911.09039*.
- de Beaudrap, N., Duncan, R., Horsman, D., & Perdrix, S. (2019). Pauli fusion: a computational model to realise quantum transformations from zx terms. *arXiv preprint arXiv:1904.12817*.
- de Beaudrap, N., & Horsman, D. (2020). The zx calculus is a language for surface code lattice surgery. *Quantum*, 4, 218.
- de Beaudrap, N., Kissinger, A., & Meichanetzidis, K. (2020). Tensor network rewriting strategies for satisfiability and counting. *arXiv preprint arXiv:2004.06455*.
- De Leon, N. P., Itoh, K. M., Kim, D., Mehta, K. K., Northup, T. E., Paik, H., ... Steuerman, D. W. (2021). Materials challenges and opportunities for quantum computing hardware. *Science*, 372(6539), eabb2823.

- Deutsch, D. E., Barenco, A., & Ekert, A. (1995). Universality in quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 449(1937), 669–677.
- Devoret, M. H., Wallraff, A., & Martinis, J. M. (2004). Superconducting qubits: A short review. *arXiv preprint cond-mat/0411174*.
- Dirac, P. A. M. (1939). A new notation for quantum mechanics. In *Mathematical proceedings of the cambridge philosophical society* (Vol. 35, pp. 416–418).
- Duncan, R., Kissinger, A., Perdrix, S., & Van De Wetering, J. (2020). Graph-theoretic simplification of quantum circuits with the zx-calculus. *Quantum*, 4, 279.
- Duncan, R., & Lucas, M. (2013). Verifying the steane code with quantomatic. *arXiv preprint arXiv:1306.4532*.
- Duncan, R., & Perdrix, S. (2009). Graph states and the necessity of euler decomposition. In *Mathematical theory and computational practice: 5th conference on computability in europe, cie 2009, heidelberg, germany, july 19-24, 2009. proceedings 5* (pp. 167–177).
- Duncan, R., & Perdrix, S. (2010). Rewriting measurement-based quantum computations with generalised flow. In *International colloquium on automata, languages, and programming* (pp. 285–296).
- Endo, S., Cai, Z., Benjamin, S. C., & Yuan, X. (2021). Hybrid quantum-classical algorithms and quantum error mitigation. *Journal of the Physical Society of Japan*, 90(3), 032001.
- Farber, R. (2011). *Cuda application design and development*. Elsevier.
- Farhi, E., Goldstone, J., & Gutmann, S. (2014). A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*.
- Fauseweh, B. (2024). Quantum many-body simulations on digital quantum computers: State-of-the-art and future challenges. *Nature Communications*,

15(1), 2123.

- Fiduccia, C. M., & Mattheyses, R. M. (1988). A linear-time heuristic for improving network partitions. In *Papers on twenty-five years of electronic design automation* (pp. 241–247).
- Forest, S., Gosset, D., Kliuchnikov, V., & McKinnon, D. (2015). Exact synthesis of single-qubit unitaries over clifford-cyclotomic gate sets. *Journal of Mathematical Physics*, 56(8).
- Fowler, A. G. (2012). Time-optimal quantum computation. *arXiv preprint arXiv:1210.4626*.
- Freedman, M., Kitaev, A., Larsen, M., & Wang, Z. (2003). Topological quantum computation. *Bulletin of the American Mathematical Society*, 40(1), 31–38.
- Fried, E. S., Sawaya, N. P., Cao, Y., Kivlichan, I. D., Romero, J., & Aspuru-Guzik, A. (2018). qtorch: The quantum tensor contraction handler. *PloS one*, 13(12), e0208510.
- Gambetta, J. M., Chow, J. M., & Steffen, M. (2017). Building logical qubits in a superconducting quantum computing system. *npj quantum information*, 3(1), 2.
- Garcia, V., Debreuve, E., & Barlaud, M. (2008). Fast k nearest neighbor search using gpu. In *2008 ieee computer society conference on computer vision and pattern recognition workshops* (pp. 1–6).
- Garvie, L., & Duncan, R. (2017). Verifying the smallest interesting colour code with quantomatic. *arXiv preprint arXiv:1706.02717*.
- Georgescu, I. (2020). Trapped ion quantum computing turns 25. *Nature Reviews Physics*, 2(6), 278–278.
- Gheorghiu, A., Kapourniotis, T., & Kashefi, E. (2019). Verification of quantum computation: An overview of existing approaches. *Theory of computing systems*, 63, 715–808.

- Gidney, C., & Fowler, A. G. (2019). Efficient magic state factories with a catalyzed $|CCZ\rangle$ to $2|T\rangle$ transformation. *Quantum*, 3, 135.
- Giles, B., & Selinger, P. (2013, March). Exact synthesis of multiqubit clifford+t circuits. *Physical Review A*, 87(3). Retrieved from <http://dx.doi.org/10.1103/PhysRevA.87.032332> doi: 10.1103/physreva.87.032332
- Gottesbüren, L., Heuer, T., Maas, N., Sanders, P., & Schlag, S. (2024). Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms*, 20(1), 1–54.
- Gottesman, D. (1998). The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*.
- Gottesman, D. (2002). An introduction to quantum error correction. In *Proceedings of symposia in applied mathematics* (Vol. 58, pp. 221–236).
- Gray, J. (2018). quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29), 819.
- Gray, J., & Chan, G. K.-L. (2024). Hyperoptimized approximate contraction of tensor networks with arbitrary geometry. *Physical Review X*, 14(1), 011009.
- Gray, J., & Kourtis, S. (2021). Hyper-optimized tensor network contraction. *Quantum*, 5, 410.
- Greenberger, D. M., Horne, M. A., & Zeilinger, A. (1989). Going beyond bell's theorem. In *Bell's theorem, quantum theory and conceptions of the universe* (pp. 69–72). Springer.
- Groote, J. F., Morel, R., Schmaltz, J., & Watkins, A. (2021). *Logic gates, circuits, processors, compilers and computers*. Springer.
- Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual acm symposium on theory of*

- computing* (pp. 212–219).
- Häffner, H., Roos, C. F., & Blatt, R. (2008). Quantum computing with trapped ions. *Physics reports*, 469(4), 155–203.
- Hanks, M., Estarellas, M. P., Munro, W. J., & Nemoto, K. (2020). Effective compression of quantum braided circuits aided by zx-calculus. *Physical Review X*, 10(4), 041030.
- Harada, T. (2007). Real-time rigid body simulation on gpus. *GPU gems*, 3, 611–632.
- Harneit, W. (2002). Fullerene-based electron-spin quantum computer. *Physical Review A*, 65(3), 032322.
- Hasan, K. S., Chatterjee, A., Radhakrishnan, S., & Antonio, J. K. (2014). Performance prediction model and analysis for compute-intensive tasks on gpus. In *Network and parallel computing: 11th ifip wg 10.3 international conference, npc 2014, ilan, taiwan, september 18-20, 2014. proceedings 11* (pp. 612–617).
- Heimendahl, A., Montealegre-Mora, F., Vallentin, F., & Gross, D. (2021). Stabilizer extent is not multiplicative. *Quantum*, 5, 400.
- Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.
- Herlihy, M., Shavit, N., Luchangco, V., & Spear, M. (2020). *The art of multiprocessor programming*. Newnes.
- Hey, T. (1999). Quantum computing: an introduction. *Computing and Control Engineering*, 10(3), 105–112.
- Hietala, K., Rand, R., Hung, S.-H., Li, L., & Hicks, M. (2020). Proving quantum programs correct. *arXiv preprint arXiv:2010.01240*.
- Hofstetter, W., & Qin, T. (2018). Quantum simulation of strongly correlated condensed matter systems. *Journal of Physics B: Atomic, Molecular and*

- Optical Physics*, 51(8), 082001.
- Huang, C., Zhang, F., Newman, M., Cai, J., Gao, X., Tian, Z., ... others (2020). Classical simulation of quantum supremacy circuits. *arXiv preprint arXiv:2005.06787*.
- Huang, H.-L., Wu, D., Fan, D., & Zhu, X. (2020). Superconducting quantum computing: a review. *Science China Information Sciences*, 63, 1–32.
- Jájá, J. (1992). *Parallel algorithms*.
- Jamadagni, A., Läuchli, A. M., & Hempel, C. (2024). Benchmarking quantum computer simulation software packages: state vector simulators. *arXiv preprint ArXiv:2401.09076*.
- Jeandel, E., Perdrix, S., & Vilmart, R. (2018). A complete axiomatisation of the zx-calculus for clifford+ t quantum mechanics. In *Proceedings of the 33rd annual acm/ieee symposium on logic in computer science* (pp. 559–568).
- Jeandel, E., Perdrix, S., & Vilmart, R. (2020). Completeness of the zx-calculus. *Logical Methods in Computer Science*, 16.
- Jo, M., & Kim, M. (2022). Simulating open quantum many-body systems using optimised circuits in digital quantum simulation. *arXiv preprint arXiv:2203.14295*.
- Kartsaklis, D., Fan, I., Yeung, R., Pearson, A., Lorenz, R., Toumi, A., ... Coecke, B. (2021). lambeq: An efficient high-level python library for quantum nlp. *arXiv preprint arXiv:2110.04236*.
- Kennes, D. M., Claassen, M., Xian, L., Georges, A., Millis, A. J., Hone, J., ... Rubio, A. (2021). Moiré heterostructures as a condensed-matter quantum simulator. *Nature Physics*, 17(2), 155–163.
- Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2), 291–307. doi: 10.1002/j.1538-7305.1970.tb01770.x

- Kim, T., Baek, K., Hwang, Y., & Bang, J. (2024). Resource-compact time-optimal quantum computation. *arXiv preprint arXiv:2405.00191*.
- Kissinger, A., & van de Wetering, J. (2019). Pyzx: Large scale automated diagrammatic reasoning [Computer software manual]. Retrieved from <https://pyzx.readthedocs.io/en/latest/api.html>
- Kissinger, A., & van de Wetering, J. (2018). Pyzx. Retrieved from <https://github.com/Quantomatic/pyzx>
- Kissinger, A., & van de Wetering, J. (2020a). PyZX: Large Scale Automated Diagrammatic Reasoning. In B. Coecke & M. Leifer (Eds.), Proceedings 16th International Conference on *quantum physics and logic*, chapman university, orange, ca, usa., 10-14 june 2019 (Vol. 318, p. 229-241). Open Publishing Association. doi: 10.4204/EPTCS.318.14
- Kissinger, A., & van de Wetering, J. (2020b, August). Reducing the number of non-clifford gates in quantum circuits. *Physical Review A*, 102(2). Retrieved from <http://dx.doi.org/10.1103/PhysRevA.102.022406> doi: 10.1103/physreva.102.022406
- Kissinger, A., & van de Wetering, J. (2021). Quizx. Retrieved from <https://github.com/Quantomatic/quizx>
- Kissinger, A., & van de Wetering, J. (2022). Simulating quantum circuits with zx-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4), 044001.
- Kissinger, A., & van de Wetering, J. (2024). *Picturing Quantum Software: An Introduction to the ZX-Calculus and Quantum Compilation*. Preprint.
- Kissinger, A., van de Wetering, J., & Vilmart, R. (2022). Classical simulation of quantum circuits with partial and graphical stabiliser decompositions. *arXiv preprint arXiv:2202.09202*.
- Kjaergaard, M., Schwartz, M. E., Braumüller, J., Krantz, P., Wang, J. I.-J., Gus-

- tavsson, S., & Oliver, W. D. (2020). Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1), 369–395.
- Kliuchnikov, V. (2013). Synthesis of unitaries with clifford+ t circuits. arxiv eprints. *arXiv preprint arXiv:1306.3200*.
- Koch, M. (2022). Quantum machine learning using the zxw-calculus. *arXiv preprint arXiv:2210.11523*.
- Koch, M., Yeung, R., & Wang, Q. (2023). Speedy contraction of zx diagrams with triangles via stabiliser decompositions. *arXiv preprint arXiv:2307.01803*.
- Koziell-Pipe, A. (2025). *Enhancing quantum computing using ai* (Unpublished doctoral dissertation). University of Oxford.
- Koziell-Pipe, A., Yeung, R., & Sutcliffe, M. (2024). Towards faster quantum circuit simulation using graph decompositions, GNNs and reinforcement learning. In *The 4th workshop on mathematical reasoning and ai at neurips'24*. Retrieved from <https://openreview.net/forum?id=54060pbCKY>
- Laakkonen, T. (2022). *Graphical stabilizer decompositions for counting problems* [Master's thesis]. Retrieved from <https://www.cs.ox.ac.uk/people/aleks.kissinger/theses/laakkonen-thesis.pdf>
- Laakkonen, T., Meichanetzidis, K., & van de Wetering, J. (2022). A graphical# sat algorithm for formulae with small clause density. *arXiv preprint arXiv:2212.08048*.
- Laakkonen, T., Meichanetzidis, K., & van de Wetering, J. (2023). Picturing counting reductions with the zh-calculus. *arXiv preprint arXiv:2304.02524*.
- Lai, J., Li, H., Tian, Z., & Zhang, Y. (2019). A multi-gpu parallel algorithm in hypersonic flow computations. *Mathematical Problems in Engineering*, 2019(1), 2053156.

- Lau, J. W. Z., Lim, K. H., Shrotriya, H., & Kwek, L. C. (2022). Nisq computing: where are we and where do we go? *AAPPS bulletin*, 32(1), 27.
- Lengauer, T. (2012). *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media.
- Li, X., Han, W., Liu, G., An, H., Xu, M., Zhou, W., & Li, Q. (2012). A speculative hmmer search implementation on gpu. In *2012 ieee 26th international parallel and distributed processing symposium workshops & phd forum* (pp. 735–741).
- Lidar, D. A., & Brun, T. A. (2013). *Quantum error correction*. Cambridge university press.
- Liu, J., Zhan, B., Wang, S., Ying, S., Liu, T., Li, Y., ... Zhan, N. (2019). Formal verification of quantum algorithms using quantum hoare logic. In *Computer aided verification: 31st international conference, cav 2019, new york city, ny, usa, july 15-18, 2019, proceedings, part ii 31* (pp. 187–207).
- Lopes, N., & Ribeiro, B. (2011). Gpumlib: An efficient open-source gpu machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3(2), 355–362.
- Mack, C. A. (2011). Fifty years of moore’s law. *IEEE Transactions on semiconductor manufacturing*, 24(2), 202–207.
- Mahadev, U. (2018). Classical verification of quantum computations. In *2018 ieee 59th annual symposium on foundations of computer science (focs)* (pp. 259–267).
- Mandrà, S., Marshall, J., Rieffel, E. G., & Biswas, R. (2021). Hybridq: A hybrid simulator for quantum circuits. In *2021 ieee/acm second international workshop on quantum computing software (qcs)* (pp. 99–109).
- Markov, I. L., & Shi, Y. (2008). Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3), 963–981.

- Martineau, M., Atkinson, P., & McIntosh-Smith, S. (2018). Benchmarking the nvidia v100 gpu and tensor cores. In *European conference on parallel processing* (pp. 444–455).
- Mazzoncini, F., Bauer, B., Brown, P., & Alléaume, R. (2023). Hybrid quantum cryptography from communication complexity. *arXiv preprint arXiv:2311.09164*.
- McAlpine, K. B. (2015). All aboard the impulse train: a retrospective analysis of the two-channel title music routine in manic miner. *The Computer Games Journal*, 4, 155–168.
- McClean, J. R., Romero, J., Babbush, R., & Aspuru-Guzik, A. (2016). The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2), 023023.
- Ng, K. F., & Wang, Q. (2018). Completeness of the zx-calculus for pure qubit clifford+ t quantum mechanics. *arXiv preprint arXiv:1801.07993*.
- Nielsen, M. A., & Chuang, I. L. (2010). *Quantum computation and quantum information*. Cambridge university press.
- Nobile, M. S., Cazzaniga, P., Tangherloni, A., & Besozzi, D. (2017). Graphics processing units in bioinformatics, computational biology and systems biology. *Briefings in bioinformatics*, 18(5), 870–885.
- NVIDIA. (2020). Cuda c++ programming guide. *NVIDIA*, July.
- Nylons, L., Harris, M., & Prins, J. (2007). Fast n-body simulation with cuda. *GPU gems*, 3, 62–66.
- Orgler, S., & Blacher, M. (2024). Optimizing tensor contraction paths: A greedy algorithm approach with improved cost functions. *arXiv preprint arXiv:2405.09644*.
- Pan, F., & Zhang, P. (2021). Simulating the sycamore quantum supremacy circuits. *arXiv preprint arXiv:2103.03074*.

- Peddie, J. (2023). *The history of the gpu-steps to invention*. Springer.
- Peres, F. C., & Galvão, E. F. (2023). Quantum circuit compilation and hybrid computation using pauli-based computation. *Quantum*, 7, 1126.
- Preskill, J. (2018). Quantum computing in the nisq era and beyond. *Quantum*, 2, 79.
- Qassim, H., Pashayan, H., & Gosset, D. (2021). Improved upper bounds on the stabilizer rank of magic states. *Quantum*, 5, 606.
- Raina, R., Madhavan, A., Ng, A. Y., et al. (2009). Large-scale deep unsupervised learning using graphics processors. In *Icml* (Vol. 9, pp. 873–880).
- Raj, S. (2022). *Graphical calculus for tensor network contractions* (Master's thesis). MA thesis, University of Oxford, 2022, visited on: 11/23/2022. 240 Victoria âl.
- Ran, S.-J., Tirrito, E., Peng, C., Chen, X., Su, G., & Lewenstein, M. (2017). Review of tensor network contraction approaches. *arXiv preprint arXiv:1708.09213*.
- Ravikumār, S. (1996). *Parallel methods for vlsi layout design*. Bloomsbury Academic. Retrieved from <https://books.google.co.uk/books?id=VPXAxkTKxXIC>
- Rebentrost, P., Mohseni, M., & Lloyd, S. (2014). Quantum support vector machine for big data classification. *Physical review letters*, 113(13), 130503.
- Ren, S., Wang, Y., & Su, X. (2022). Hybrid quantum key distribution network. *Science China Information Sciences*, 65(10), 200502.
- Rendl, F., & Sotirov, R. (2018). The min-cut and vertex separator problem. *Computational Optimization and Applications*, 69(1), 159–187.
- Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 120–126.

- Roffe, J. (2019). Quantum error correction: an introductory guide. *Contemporary Physics*, 60(3), 226–245.
- Rutenbar, R. A. (1989). Simulated annealing algorithms: An overview. *IEEE Circuits and Devices magazine*, 5(1), 19–26.
- Sander, A., Burgholzer, L., & Wille, R. (2024). Equivalence checking of quantum circuits via intermediary matrix product operator. *arXiv preprint arXiv:2410.10946*.
- Sanders, J., & Kandrot, E. (2010). *Cuda by example: an introduction to general-purpose gpu programming*. Addison-Wesley Professional.
- Scarani, V., Bechmann-Pasquinucci, H., Cerf, N. J., Dušek, M., Lütkenhaus, N., & Peev, M. (2009). The security of practical quantum key distribution. *Reviews of modern physics*, 81(3), 1301–1350.
- Schaller, R. R. (1997). Moore’s law: past, present and future. *IEEE spectrum*, 34(6), 52–59.
- Schlag, S. (2020). *High-quality hypergraph partitioning* (PhD thesis). Karlsruhe Institute of Technology, Germany.
- Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., & Sanders, P. (2022, mar). High-quality hypergraph partitioning. *ACM J. Exp. Algorithmics*. Retrieved from <https://doi.org/10.1145/3529090> doi: 10.1145/3529090
- Schwetz, M., & Noack, R. (2024). Three-qubit deutsch–jozsa in measurement-based quantum computing. *International Journal of Quantum Information*, 22(07), 2350046.
- Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2), 303–332.
- Slussarenko, S., & Pryde, G. J. (2019). Photonic quantum information processing: A concise review. *Applied physics reviews*, 6(4).

- Stallings, W. (2010). *Computer organization and architecture: Designing for performance 8th edition*. Pearson Education, Inc.,.
- Stollenwerk, T., & Hadfield, S. (2024). Measurement-based quantum approximate optimization. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 1115–1127).
- Sutcliffe, M. (2024a). *Paramzx*. Retrieved from <https://github.com/mjsutcliffe99/ParamZX>
- Sutcliffe, M. (2024b). *Procoptcut*. Retrieved from <https://github.com/mjsutcliffe99/ProcOptCut>
- Sutcliffe, M. (2024c). Smarter k-partitioning of zx-diagrams for improved quantum circuit simulation. *arXiv preprint arXiv:2409.00828*.
- Sutcliffe, M. (2024d). *Zx-partitioner*. Retrieved from <https://github.com/mjsutcliffe99/zxpartitioner>
- Sutcliffe, M., & Kissinger, A. (2024a). Fast classical simulation of quantum circuits via parametric rewriting in the zx-calculus. *arXiv preprint arXiv:2403.06777*.
- Sutcliffe, M., & Kissinger, A. (2024b, August). Procedurally optimised zx-diagram cutting for efficient t-decomposition in classical simulation. *Electronic Proceedings in Theoretical Computer Science*, 406, 63–78. Retrieved from <http://dx.doi.org/10.4204/EPTCS.406.3> doi: 10.4204/eptcs.406.3
- Takeda, S., & Furusawa, A. (2019). Toward large-scale fault-tolerant universal photonic quantum computing. *APL Photonics*, 4(6).
- Theis, T. N., & Wong, H.-S. P. (2017). The end of moore’s law: A new beginning for information technology. *Computing in science & engineering*, 19(2), 41–50.
- Toumi, A. (2022). *Category theory for quantum natural language pro-*

- cessing* (Doctoral dissertation, University of Oxford). Retrieved from [arXivpreprintarXiv:2212.06615](https://arxiv.org/abs/2212.06615)
- Toumi, A., Yeung, R., & de Felice, G. (2021). Diagrammatic differentiation for quantum machine learning. *arXiv preprint arXiv:2103.07960*.
- Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363), 5.
- van de Wetering, J. (2020). Zx-calculus for the working quantum computer scientist. *arXiv preprint arXiv:2012.13966*.
- Van Laarhoven, P. J., Aarts, E. H., van Laarhoven, P. J., & Aarts, E. H. (1987). *Simulated annealing: Theory and applications*. Springer.
- Vidal, G. (2003). Efficient classical simulation of slightly entangled quantum computations. *Physical review letters*, 91(14), 147902.
- Vollmeier, Y. (2025). Graphical stabilizer decompositions for multi-control toffoli gate dense quantum circuits. *High school thesis*. Retrieved from <https://arxiv.org/abs/2503.03798>
- Wahl, T. B., & Strelchuk, S. (2023). Simulating quantum circuits using efficient tensor network contraction algorithms with subexponential upper bound. *Physical Review Letters*, 131(18), 180601.
- Wang, J., Sciarrino, F., Laing, A., & Thompson, M. G. (2020). Integrated photonic quantum technologies. *Nature Photonics*, 14(5), 273–284.
- Wang, Q. (2022). Completeness of the zx-calculus. *arXiv preprint arXiv:2209.14894*.
- Wang, Q., & Yeung, R. (2022). Differentiating and integrating zx diagrams. *arXiv preprint arXiv:2201.13250*.
- Wang, Q., Yeung, R., & Koch, M. (2024). Differentiating and integrating zx diagrams with applications to quantum machine learning. *Quantum*, 8, 1491.
- Wesenberg, J. H., Ardavan, A., Briggs, G. A. D., Morton, J. J., Schoelkopf, R. J.,

- Schuster, D. I., & Mølmer, K. (2009). Quantum computing with an electron spin ensemble. *Physical Review Letters*, 103(7), 070502.
- Whalen, S. (2005). Audio and the graphics processing unit. *Author report, University of California Davis*, 47, 51.
- Wootters, W. K., & Zurek, W. H. (1982). A single quantum cannot be cloned. *Nature*, 299(5886), 802–803.
- Xu, X., Benjamin, S., Sun, J., Yuan, X., & Zhang, P. (2023). A herculean task: Classical simulation of quantum computers. *arXiv preprint arXiv:2302.08880*.
- Yeung, R. (2020). Diagrammatic design and study of ansatze for quantum machine learning. *arXiv preprint arXiv:2011.11073*.
- Young, K., Scese, M., & Ebneenasir, A. (2023). Simulating quantum computations on classical machines: A survey. *arXiv preprint arXiv:2311.16505*.
- Zaman, K., Ahmed, T., Hanif, M. A., Marchisio, A., & Shafique, M. (2024). A comparative analysis of hybrid-quantum classical neural networks. *arXiv preprint arXiv:2402.10540*.
- Zhang, S.-X., Wan, Z.-Q., Lee, C.-K., Hsieh, C.-Y., Zhang, S., & Yao, H. (2022). Variational quantum-neural hybrid eigensolver. *Physical Review Letters*, 128(12), 120502.
- Zhang, Y., Zhang, Y., Portokalidis, G., & Xu, J. (2022). Towards understanding the runtime performance of rust. In *Proceedings of the 37th ieee/acm international conference on automated software engineering* (pp. 1–6).
- Zulehner, A., & Wille, R. (2018). Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), 848–859.
- Zurek, W. H. (2003). Decoherence, einselection, and the quantum origins of the classical. *Reviews of modern physics*, 75(3), 715.