**Parallelised repeated classical simulation of quantum circuits via generalisation of the ZX-calculus**

**Matthew Sutcliffe, Aleks Kissinger**

Department of Computer Science, University of Oxford, Oxford, UK

2023

The basis of this project was recognising that, in many tasks involving ZX-calculus reductions over many diagrams, much of the computational work is repeated (albeit with different exact numbers involved). From this standpoint, it was shown that many similar reductions can instead be formulated as a single reduction, while maintaining generality via parameterisation. Specifically, this paper explores how this reasoning can be applied to show how one can arrive at a generalised scalar result for simplifying a Clifford+T circuit, for any arbitrary Boolean inputs. To take many repeat results of simplifying such circuits for various inputs then, it was shown that, given the generalised scalar, this can be attained by taking many evaluations of this result, instead of undergoing many (slower) circuit reductions. Lastly, this paper covers how these repeated evaluations can be computed very efficiently and in parallel – in a way that would be largely impractical for direct circuit reduction – utilising the GPU. Using these methods, speedups upwards of a 100x were achieved for repeated classical simulations of Clifford+T circuits, versus the direct ZX-based reduction approach.

**1. Introduction**

ZX-calculus is a powerful formulation for expressing quantum circuits and subsequently reducing them to simpler forms in a very efficient and tidy manner. Indeed, scalar circuits (those with given inputs and outputs), can be reduced efficiently in this way to a final numerical (scalar) result. One application where this is relevant is in classical simulation of quantum circuits, which is particularly useful when access to sufficiently powerful quantum hardware (perhaps beyond even those available today) is scarce. However, without the benefits of quantum computation, simulating such circuits is a very slow task for classical hardware, with runtimes scaling exponentially with the complexity of the circuit.

Furthermore, as quantum circuits are inherently non-deterministic, one may observe different results every time the circuit is run (albeit in accordance with some – typically unknown – probability distribution). Consequently, it can be desirable to run such circuits not just once, but a number of times (particularly if the algorithm is specifically intended to feature some degree of randomness in its output).

[TODO – discuss the precursor paper(s)]

The subsequent work of Kissinger and van de Wetering [REF], showed how applying ZX-calculus reductions at every step of the decomposition leads to a significant reduction in the number of resulting diagrams, by allowing the costly T-gates to cancel wherever possible. With their approach, they were able to simulate 50-qubit circuits of up to about 40 T-gates consistently within 5 minutes (and beyond 50 T-gates for 100-qubit circuits).

With this method and others, however, classically simulating a given circuit a second time requires repeating the whole process and hence doubling the overall runtime. In this paper, a generalisation of the work of Kissinger and van de Wetering is introduced, allowing for circuits to be reduced without prior enumeration of its inputs. Ultimately, this allows – once the circuit has been reduced for the generalised case – for the respective results of repeated sampling (for various inputs) to be calculated very rapidly, by simply evaluating the generalised scalar. Indeed, with the generalised scalar expressed appropriately, these evaluations can be calculated in parallel on the GPU. Consequently, as the chapters ahead will show, after initial generalised reductions, every subsequent sampling of classical simulation for a given circuit can be computed in a small fraction of the time that would be achieved by repeating the whole process.

[TODO]

**2. Background**

*2.1. ZX-calculus*

Quantum algorithms are commonly expressed graphically in the form of quantum circuit diagrams [1], which show the sequence of gates acting upon the qubits. A useful alternative notation, however, is that of ZX-calculus, wherein quantum circuits are expressed as *ZX-diagrams* [2]. These are graphs which denote phase gates as coloured vertices (or *spiders*) and their connections as edges (or *wires*, or *legs*) between these vertices. There are two types of spiders, namely *Z* (typically green or white) spiders, which denote rotations about the Z-axis of the Bloch sphere, and *X* (typically red or grey) spiders, which denote rotations about the X-axis of the Bloch sphere. Each spider has also an associated *phase* (relating to the angle of rotation it induces), written either within the spider or alongside it (or ignored if zero). The Hadamard gate is also included, expressed as a box of typically yellow or white, though this may also be decomposed into *Z* and *X* spiders in accordance with the Euler decomposition rule (more on rewrite rules shortly) [2]. An edge containing a Hadamard gate may also instead be expressed as a dotted or blue line, known as a Hadamard edge. The result is a graph of such connected spiders whose edges may be arbitrarily deformed and whose layout arbitrarily rearranged. In other words, as is the first meta-rule of ZX-calculus: only connectivity matters.

ZX-calculus is logically complete [3] and thus any quantum circuit can be expressed in this manner and benefit from its advantages. These advantages are plentiful. In addition to the visually comprehensible design and free-form lack of structure offered by the calculus, it is also equipped with a number of *rewrite rules* that describe how certain structures within a ZX-diagram may be re-expressed as an equivalent (generally simplified) structure. Figure 2.1 shows the basic elementary rewrite rules. It should be noted that all rewrite rules still apply with all its spider colours inverted.

[FIG 2.1 HERE [REF WORKING SCIENTIST]. Caption: A set of the basic rewrite rules of ZX-calculus, where Greek letters denote arbitrary real variables, , and Latin letters denote arbitrary Boolean variables, . Note that the rules still apply if all the spider colours are inverted.]

From this set, there are also a number of *derived* rules. Two of particular interest in this paper are the *local complementation* and *pivoting* rules, shown in figure 2.2. For a more comprehensive list of the rewrite rules, see *ZX-calculus for the working computer scientist* [REF].

[FIG 2.2]

To demonstrate the power of ZX-calculus, figure 2.3 shows an example quantum circuit and its ZX-diagram representation, along with the result of simplifying this circuit through the use of various rewrite rules. In this fashion, a rather complex quantum circuit may be simplified to an equivalent circuit of fewer gates and minimised complexity. More than just a superficial improvement, this allows otherwise computationally costly circuits to be executed on the limited-scale quantum computers of today with reduced time and noise.

[FIG 2.3]

[TODO: Discuss briefly T-gates, and Clifford vs Clifford+T]

*2.2. Stabiliser decompositions*

[TODO]

**3. Method**

*3.1. Generalising ZX-calculus*

Software implementations of ZX-calculus (namely the Python-based *PyZX* [REF] and the Rust-based *Quizx* [REF]) do not at present generally support parameterised phases. That is to say every spider requires an exact numerical phase (such as , , or ), rather than some undefined symbolic variable (such as or ). The reason for this is that – as can be seen in figures 2.1 and 2.2 – many of the rewrite rules require one or more of its spiders’ phases to take some specific value, or some value within a specific set. Take, for example, the *state copy* rule (see figure 2.1). If both spiders on the left-hand side take unknown phases, and , then it remains unknown whether the rule may apply as it is undefined whether as required. As such, it must be assumed that the rule does not apply, since it does not *necessarily* do so. Likewise, the *local complementation* rule requires a spider of phase and thus given an undefined symbolic phase, such as , it would assume that the rule does not apply. While this same reasoning is not true of *all* the rules (*spider fusion*, for example, can always straightforwardly apply without any care for the phases involved), it is true of enough to prevent much practical simplification of parameterised (symbolically phased) graphs.

While this would be true of a *fully* generalised implementation of ZX-calculus, allowing any arbitrary parameterised phases (i.e. ), if instead the situation is appropriately restricted it can yield some useful results. In particular, this paper considers the case wherein all parameters are limited to Boolean multiples of (i.e. ). This applies also to every parameterised *term*, meaning a phase of is allowed but is not. Additionally, the gate-set is initially restricted to Cliffords, meaning all phases are bound to . Under these restrictions, the rewrite rules can (almost) all be maintained without loss of generality. (One notable exception is the *identity removal* rule, which insists on a zero-phase spider, thus remaining ambiguous for a parameterised phase, . However, this rule is also exceptional in being a largely trivial one to the concerns here as they can be pushed aside and largely ignored.)

In this regime, any ambiguity in a phase is always by a term of . And as amounts to half a rotation, any parameterised phase will always have two possible (and necessarily opposite) states. For instance, a phase of may take only a value of or (i.e. ); a phase of may also take only a of value or (i.e. ); and a phase of may take only a value of or (i.e. ). Referencing the phase wheel shown in figure 3.1, the two possible states of any parameterised phase are thus always an opposite pair among the Clifford options (i.e. or ). Herein, the possible states a particular parameterised phase may take will be referred to as its *domain*. Note also that any phase will thus necessarily take the form of a sum of Boolean parameters (or none) plus some Clifford constant: .

|  |  |  |
| --- | --- | --- |
|  |  |  |

[FIG 3.1 – add caption]

It is due to the fact that the domain of any phase is always two opposite values on this wheel (i.e. always separated by ) that the ambiguity remains a stable and undisruptive one to the rewrite rules. Consider again the *local complementation* rule. While the central spider may take a parameterised (and hence ambiguous) phase, it will nevertheless always be known whether this rule applies. Either its domain is , meaning the rule definitely does not apply, or it is , meaning it definitely does. If indeed the rule does apply, then the ambiguity in this phase simply propagates through the rule. In the case of *local complementation* this becomes generalised as in figure 3.2.

[FIG 3.2, showing local comp, but with (a+b+…)+-pi/2 -> etc. **ANOTHER WAY TO LOOK AT IT IS A -> A+PI**]

Most of the other rewrite rules can likewise be generalised quite straightforwardly by the same reasoning, as any effect they have on phase amounts to adding (or, equivalently, subtracting) . The notable exception is the *-commutation rule*, as this triggers a phase change of (rather than, say, ). This can be easily resolved, however, by separating the rule into two distinct rules, outlined in figure 3.3.

[FIG 3.3 – mention also these apply respectively to the vertical and horizontal axes of the phase wheel. Mention also the role of the identity removal rule in these rules]

Having now justified that ZX-Calculus remains logically complete for Clifford circuits after introducing Boolean parameters, it becomes worthwhile considering the impact of extending the gate-set to that of Clifford+T circuits. In this regime, every phase is given by the sum of some parameters (or none) plus some constant , where, as before, each parameter term is strictly Boolean: (e.g. is disallowed) [CALL THIS RULE A or something?].

After extending the gate-set to phases, the same logic as above can be applied to show that most of the rewrite rules can still remain generalised. For a given phase, such as , one only needs to take note of the constant term to know whether or not a given rule applies. In this example, it is known that this phase can only equal or , depending on the values of and (which can each be 0 or 1). As such, its domain remains clearly known – corresponding to an opposite pair on the phase wheel (see figure 3.1). Consequently, it is known that – for this phase – the local complementation rule, for instance, does not apply (as it requires a phase in the domain ).

The inconvenient exception, however, (aside from the trivial identity removal rule) is again the -commutation rule. To generalise this rule, it would have to be further separated into 4 distinct cases (applying to the different options of domain for the central spider’s phase), as outlined in figure 3.4. The problem here is that, given a non-Clifford phase in the central spider, the difference in the result of applying the rule for a phase given one set of parameter values versus another cannot be expressed generally with a conditional phase. For instance, if the central phase is then applying the rule would need to map this to if and if . However, both of these mappings correspond to a change of a phase of , and as such would therefore necessarily have to be generalised for parameters as: . Rather problematically, this violates RULE A, by introducing *fractional* parameter terms into a phase (in this case ).

[FIG 3.4]

To see why this is a problem, consider a spider of phase . The domain of this phase is no longer confined to two opposite points on the phase wheel. In fact, in this case, its domain includes 4 possible states: , depending on the values of its parameters. Consequently, one cannot say with certainty whether, for example, the local complementation rule may apply. Thus, the rule cannot be neatly generalised in a way that maintains consistent graph structure. One may thus conclude that ZX-calculus can be generalised in this fashion for Clifford circuits, but not for the wider Clifford+T set.

There is, however, a silver lining when applying this method to the Clifford+T set. Specifically, when dealing with scalar diagrams – that is, those with plugged inputs and outputs – the aim of simplification is typically to reduce the diagram fully to the complex scalar it represents. The rewrite strategies that achieve this typically do so via decomposition of the diagram’s T-gates into a sum of Cliffords, as outlined in section 2.2. This means that Clifford+T scalar diagrams can be reduced to their final scalar form using only rewrite rules on Cliffords. In the simplest case, this would mean decomposing all T-gates in the original Clifford+T diagram to produce a large sum of Clifford diagrams which could then each be simplified to their scalar form, without ever needing to apply any rewrite rules to a T-like phase. This would allow the generalisation of parameterised ZX-calculus outlined above to be utilised without running into the issues that arise when T-like phases meet the -commutation rule.

Better yet, one can proceed as normal with the Clifford+T scalar diagram reduction strategies – undergoing simplification on Clifford+T diagrams at each stage – *as far as possible*, while maintaining generality and simply neglecting the disruptive -commutation rule. Indeed, the particular strategy this paper considers is that of [REF; Also outlined above in section 2 somewhere?], which primarily relies on *local complementation* and *pivoting* (both of which – as has been shown above – remain reliable when generalised even for Clifford+T diagrams). [Indeed, this rule hardly even comes into the specific simplification strategy we are considering, so while it technically can result in more summands at the end, realistically it seldom does?]

This strategy, therefore, amounts to iteratively applying the following set of rules, shown in figure 3.5, in conjunction with the T-gate decomposition outlined in section 2.2[?] when no further simplification is possible with the former rules. It is important to note that these two rules have been generalised to support parameterised spiders and these parameters are appropriately absorbed into the scalars when the rules are applied. The derivations of these parameterised versions of these rules follow in DERIV 3.1[?].

[FIG 3.5. See page 8 of Kissinger/Wetering paper, but add param to scalars]

[DERIV 3.1?? Explain how etc works. i.e. applies e^ipi/4 if (a+b)=1 and applies e^-ipi/4 if (a+b)=0 or whatever]

Moreover, for reasons that will become apparent, the parameterised components of the scalar are isolated from the constant parts. For instance, applying the local complementation rule to a central spider of phase not only alters the ZX-diagram but also introduces a new scalar coefficient of [where n is…]. This can be seen as the product of a parameterised component, , and a constant component, . The constant part can be simply multiplied into the global scalar term of the ZX-diagram (which may be one among many summands, due to the T-gate decompositions). (Note, however, that it would RETAIN UN-EVALUATED FORM, I.E. EXP(I\*PI\*SOME\_TALLY\_HERE) AND SQRT(2)^POWER, ETC.) The *parameterised* component, meanwhile, may be added to a list of like components that apply to the present diagram. So, if the local complementation rule is collectively applied 4 times for various parameterised vertices in the diagram, then, along with simplifying the diagram and altering its constant scalar, it will also introduce 4 parameterised scalar factors, such as .

Each such parameterised scalar factor may be denoted as a *subterm*, while each individual scalar summand (reduced to a scalar from a ZX-diagram) – composed of the product of a number of subterms (including one constant factor subterm) – may be denoted a *term*. In summary, with this terminology, one takes some initial scalar Clifford+T ZX-diagram and, via decompositions of the T-gates, expresses it as the sum of scalar Clifford ZX-diagrams. After exhaustively simplifying these diagrams, each reduces to simply a scalar *term*, being the product of parameterised (and one constant) *subterms*. So, each term is given by the product of its constituent subterms, and the final result is given by the sum of each term.

In the examples above, it was shown that subterms of the form arise whenever the local complementation rule is applied to a parameterised vertex. This is, however, just one type of subterm, which we may refer to as an *lcomp subterm* or a *half-pi subterm*. A small number of other distinct types of subterms may also arise; namely *node subterms*, *spider-pair* (or *phase-pair*) *subterms*, and lastly *pivot* (or *pi-pair*) *subterms*. These are summarised in table 3.1, highlighting their respective forms and origins.

[Maybe justify/derive each one before we give the summary table]

[TABLE 3.1 – Make this table look nicer – and FINISH IT, and give it a caption, etc. Make a note that C is a Clifford+T const (n/4) and B is a Boolean const (0 or 1). Note that we haven’t yet explained why lcomp type is ½ or 3/2 and not 2/2 – which becomes a pi-pair term??] Also, clarify that in halfpi type, you cannot simply expand brackets.[Maybe also change here, AND EVERYWHERE ELSE, e^(i\*pi\*C xor a xor b) with e^(i\*pi\*(C PLUS a xor b)) etc.]

|  |  |  |
| --- | --- | --- |
| **Name of Type** | **Form** | **Origin** |
| Node |  | Arise from standalone legless spiders |
| Spider-pair / phase-pair |  | Define A and B here |
| lcomp / half-pi |  | Arise from applying the local complementation rule to a parameterised vertex |
| Pivot / pi-pair |  | Arise from applying the pivoting rule to a pair of vertices (of which at least one is parameterised) |

What this amounts to is that any parameterised Clifford circuit may be reduced to a scalar term, composed of the product of some amount of such subterms (as well as some constant factor). In turn, any Clifford+T circuit may be reduced to a sum of such terms. This is illustrated in figure 3.6. Note that each term may have a different number of subterms (and these subterms themselves may be very different from one term to the next). Further note that if the original ZX-diagram were a Clifford one (as opposed to Clifford+T), then it would be described by just one term, made up of a number of subterms. As a brief example, a Clifford+T circuit with parameterised inputs may be reduced to the generalised scalar form: . [Maybe show the generic case somewhere too: EQUATION X: i.e. SUM[PROD(node\_type\_n)\*PROD(next\_type\_n),etc.]

|  |  |  |
| --- | --- | --- |
|  |  |  |

[FIG 3.6 below. Caption it and tidy it.]

|  |  |  |
| --- | --- | --- |
|  |  |  |

To arrive at the final result of the scalar, one then need only substitute in values for the parameters () and evaluate the resulting terms. As a simple Clifford example, consider the ZX-diagram of figure 3.7 with its parameterised inputs, . Supposing one wished to deduce the scalar this diagram represents for a given input, such as , traditionally they would substitute in these values ( and then simplify the resulting (now non-parameterised) diagram. In the alternative approach offered by this paper, one could instead leave the original diagram in its parameterised form and simplify it to a parameterised scalar term (being the product of such subterms as in table 3.1) while maintaining generality. Only then would they be required to substitute in the desired parameter values ( and evaluate the final result.

[FIG 3.7. (a) A simple scalar Clifford with inputs a,b,c,d,e, and (b) its reduced parameterised scalar form (being the product of subterms of table 3.1, and some const.)]

Now suppose one wished to find the scalar result of not just one but every (or at least many) possible combination of input values, for input states , for the diagram of figure 3.7. As this diagram takes 5 inputs (parameterised as ), and each has two possible states ( or ), then there are possible combinations of input values. In the traditional approach, this would require substituting in the values and simplifying the whole diagram 32 times to find all 32 results. In this alternative (parameterised) approach, it would require simplifying the original diagram just *once*, and subsequently evaluating the resulting parameterised scalar expression 32 times (once for each combination of parameter values).

At this stage, little (if any) speedup in computational runtime is offered, as evaluating some long complex expression, such as [TODO – from fig 3.7], is comparably computationally costly as just simplifying the original diagram (hence 32 *evaluations* of the resulting parameterised scalar is roughly comparable in speed with 32 *simplifications* of the original diagram). The opportunity for speedup arises from the fact that, having reduced the diagram to a generalised (parameterised) scalar expression – whose terms all share the same restrictive set of forms/structures – it is now in a highly parallelisable state and requiring only very basic computation. That is to say, given the resulting parameterised expression denoting the scalar form of the diagram, all its possible evaluations for the various combinations of parameter values can be computed in parallel, and only requiring very computationally cheap basic mathematical operations. Consequently, it is now in a form that lends itself very nicely to be processed on the GPU rather than the CPU.

[Maybe a subsection here on simplifications we can do to the final scalar expression, such as cancelling opposite like terms, etc.]

*3.2. Parallel evaluations on the GPU*

With the data now in a highly parallelisable form, it remains to be structured in a computationally concise way that lends itself to easy evaluation. Hitherto, the structure of the data has made use of relatively high-level features, such as *dicts* and *sets*, as these are very useful for efficiently performing, for instance, a symmetric difference operation (effectively an XOR) between two sets of parameters. However, from this point, it becomes necessary to store the scalar terms in a more primitive data structure, both to minimise the space and complexity of the data before sending it to the GPU, and to accommodate the fact that the GPU code will be C-based and is benefitted from remaining as simple as possible.

As such, from this point, the data structure that will be used to represent the scalar terms will be a simple set of two-dimensional arrays – one for each subterm type. Within each of these 2D arrays, each row will record one of the subterms of that type, by noting which parameters and constants are involved (taking the constants as coefficients of , such that they are integers in the range ). In this way, node-type subterms can be recorded by simply flagging which parameters are included in its exponential, as well as noting its constant factor. Consider, for example, within a system of 4 parameters, a term consisting of two node-type subterms, , among other subterm types (which can be ignored for now). The node-type subterm data for this term may be represented as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C | α | β | γ | δ |
| 2 | 1 | 1 | 0 | 0 |
| 4 | 0 | 1 | 1 | 1 |

Furthermore, to maximise parallelism, for each subterm type, it is ideal to have the same number of such subterms within each term. As this is not naturally the case, it will be artificially enforced padding out all terms with dummy subterms (that effectively act as a multiplication by unity) such that all are of equal length, for each type of subterm. For instance, consider a scalar that is made up of 3 terms – with 2, 1, and 3 node-type subterms respectively. The first term could be padded with one extra dummy node-type subterm, and likewise the second term could be padded with two extra such dummy subterms. This way all 3 terms will have the same number of node-type subterms (namely 3).

With padding to ensure all terms have the same number of node-type subterms, the rows of node-type data for each respective term may be stacked into one array, as in the following example of 3 terms (respectively with 2, 1, and 3 node-type subterms). Note that an additional column is included to the left. This records whether the row is an actual subterm to be processed (in which case this column is set to 1) or a dummy subterm added for padding (set to 0) which can be ignored. [Also mention that each cell is an int – with a bracket to mention that the use of bits instead of bytes is discussed in a later section]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | C | α | β | γ | δ |
| 1 | 2 | 1 | 1 | 0 | 0 |
| 1 | 4 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 6 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 5 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 2 | 0 | 0 | 1 | 0 |

There is an important note to be made here regarding how this data is to be stored in memory. While for practical purposes this data is treated as two-dimensional, as far as the memory is concerned it is in fact, necessarily stored linearly. Conventionally, this would be stored, linearised, in row-major order (storing the cells of the first row, followed by those of the second row and so on). This is appropriate when the data is to be processed row-by-row on the CPU (whose processing pattern is inherently sequential), as each subsequent element of data to be processed is stored immediately after the previous, ensuring convenient (quicker) access.

However, if instead, as in this case, each row of this data is to be processed in parallel on the GPU, then this memory arrangement proves to be suboptimal. In this scenario, it is preferential to store the data in column-major order (meaning all the cells of the first column are stored one after the other, followed then by the cells of the second column and so on). This is because when the rows are to be processed in parallel, they will process their respective *multiplier* factors (their data of the first column) at the same time, and then their respective *constant* factors (the second column) at the same time, and so on. Consequently, storing all the *multiplier* factors (the first column) together in memory means that they can all be retrieved very efficiently, with many being moved in one go (as opposed to individually locating and sending each row’s multiplier factor one by one). For this reason, the two-dimensional data of the node-type subterms (and likewise for the 2D arrays of other subterm types) will be stored in column-major order, for a far more efficient memory access pattern.

In a similar fashion to the node data, all the half-pi subterms of the parameterised scalar can be represented with rows of a 2D array, taking the following columns. As before, the first column signifies whether the row is a genuine entry to be processed or simply a dummy row for padding. In the latter case, it will simply be ignored. Also as before, this example is for a scalar of 4 parameters (), but this can be applied for a scalar any arbitrary number of parameters (giving columns).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | B | α | β | γ | δ |

Similarly, the pi-pair type subterms can be recorded using the following columns (for parameters). The first column is again a dummy flag. Meanwhile, the constant column and set of parameter columns record the data in the first bracket of the subterm, while the second set records the data in the second bracket (see table 3.1 for the formulaic expression of half-pi subterms).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* |  |  |  |  |  |  |  |  |  |  |

Lastly, phase-pair type subterms can be recorded using the same column headers as pi-pair types (though notably they are nevertheless kept in their own separate dataset).

So, now the data for the parameterised scalar is (almost) fully recorded in 4 separate 2D arrays. An additional final array is required to record the constant factors of each term. As such, this final array has considerably fewer rows, as they each represent one of the terms of the scalar rather than one of the subterms. Each of these constant factors is a complex number, though can be expressed neatly (and fairly precisely) in the form: , where , , and . Given this, the constant factor of each term can be recorded as a row in the following, where and are integers and and are floats.

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

In addition to these 5 two-dimensional arrays, a few extra standalone integers are needed to record the relevant metadata – namely (a) the number of parameters, (b) the number of terms, (c) the number of node-type subterms in each term (remember this is the same for all of them, as they have all been padded to match the number for the maximum case), (d) the number of half-pi subterms in each term, (e) the number of pi-pair subterms in each term, and lastly (f) the number of phase-pair subterms in each term.

Altogether, this data completely describes a parameterised scalar of the form of [EQUATION X – see earlier], in a concise manner that is efficiently structured for GPU processing. Note that for a scalar of parameters, the amount of memory this takes is , where [DEFINE TERMS, e.g. S\_int = integer size (one byte char actually) = 1 byte, etc.] [Note that in section whatever, the prospect of packing bytes is discussed to minimise space further]. [Maybe write a bit about approx. sizes and how much becomes too much for GPU and mention that we have ways around this, mentioned later]

With the data now packaged in this way, it is ready to be sent to the GPU for processing. Thereupon, it is to be evaluated based on some set of parameter values (e.g. ) that are also to be sent to the GPU (though this amounts to a trivially small bytes [see later chapter for bits vs bytes discussion] of data, given parameters). Along with two fixed cells of value 1 (to correspond to the multiplier and constant factor columns of the subterm datasets), these parameter values can be treated as a one-dimensional array of length , such as:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | C | α | β | γ | δ |
| 1 | 1 | 1 | 0 | 1 | 0 |

Evaluating the node-type subterms first, the node-type data will be loaded to a kernel, with each row processed on its own thread (meaning the rows are processed in parallel\*[note: not all, since there is a limit on the thread count, blah blah blah]). With a quick initial check, any subterms whose first cell is 0 (that is any dummy subterm) can immediately return 1 from the kernel without needing to undergo any further calculation. For the remaining (non-dummy) subterms, the first step it will undergo is to multiply each row (in parallel) by the above row of parameter values. Using the same example as before, this produces the following result:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | C | α | β | γ | δ |
| 1 | 1 | 1 | 0 | 1 | 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | C | α | β | γ | δ |
| 1 | 2 | 1 | 1 | 0 | 0 |
| 1 | 4 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 6 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 5 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 2 | 0 | 0 | 1 | 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* | C | α | β | γ | δ |
| 1 | 2 | 1 | 0 | 0 | 0 |
| 1 | 4 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 6 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 5 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 2 | 0 | 0 | 1 | 0 |

This amounts to substituting in all the parameter values wherever they appear in the subterm expressions. Essentially, this means each row has been taken from a parameterised form, such as , to an unsimplified constant form, such as . The next step is the reduce the string of XOR’d Booleans in the exponent. This is done, for each row in parallel, by simply summing all the cells of the row (except the first two) and taking the modulo 2 (hence, 1 if odd and 0 if even) of the result. In the example case, this effectively takes the expression to , with all the XOR’d exponent terms (except the original constant one) having been reduced to either 0 or 1. [Maybe somewhere, much earlier on, write a brief note that a xor b xor c = (a+b+c)%2]. From here, the exponent can be reduced fully by adding this XOR sum result (being either 0 or 1) to the constant factor and taking again the modulus 2. In the example, this essentially results in arriving at .

Now, the subterm can be simplified. As calculations involving complex exponentials are fairly costly, and since there are only 8 possible solutions for each node-type subterm, a lookup table can be used to map each exponent value (which have just been calculated) to a simplified result. Specifically, this means checking the exponent value that has now been arrived at and based on this value returning the appropriate result among the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | | |
| *A* | *B* | *C* | *D* |
| 0 | 2 | 0 | 0 | 0 |
| 1 | 1 | 1/2 | 0 | 1/2 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 1 | -1/2 | 0 | 1/2 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 1 | -1/2 | 0 | -1/2 |
| 6 | 1 | 0 | -1 | 0 |
| 7 | 1 | 1/2 | 0 | -1/2 |

[rename the “multiplier flag”; don’t call it the multiplier flag anywhere, as that is a misleading name]

Note that each resulting expression is stored using 4 real numbers, taking the form: . In the example subterm, , the exponent is , meaning it is mapped in the lookup table above according to , returning the expression . More specifically, it returns the values .

At this point, from each node-type subterm (that is, each row of the node-type data) a resulting set of 4 numbers has been produced, each relating to one of the expressions in the right set of columns above (and with any dummy subterms returning simply to act as an inert multiplier of 1).

[do likewise for the other term types – no need to go into as much detail for the others though. But show pseucodoe for all the kernels somewhere maybe?] [also, don’t forget to include the constant factors of each term, and not just its subterms] [also mention somewhere that you take copies of the data or whatever so as to keep the original in tact on the gpu for the next sampling]

Having now evaluated all the subterms, the penultimate step is, for each term, to multiply together all its constituent subterms. From here, each thread can be assigned to one term (instead of each thread being assigned to one subterm), and so each term can, in parallel, simply compute the product of its subterms by iteratively multiplying each one by the next. (Remember, the GPU has already been told how many subterms, including the dummies, are in each term – with this number being the same for all terms.) To again ease the calculations and to avoid dealing explicitly with multiplying complex numbers, the following relationships can be utilised to multiply each expression to the term’s running total:

[TODO; the “a -> b + c”, “b -> -a +d” or whatever things]

(Meanwhile, adding such expressions, which will be necessary in the next step, is as simple as adding the like coefficients.)

Now, the GPU has arrived at a set of evaluated and simplified constant terms – each of the form . All that remains is to sum them together. To make one final use of GPU parallelism, the following algorithm is utilised to achieve this in iterations [?] [assuming there are not more than twice as many terms as threads??] (rather than n iterations as would be achieved by naïvely summing each term to the total in sequence), for terms.

[PSEUDOCODE HERE (or in appendix?) OF THE SUMMATION ALGORITHM]

At last, the GPU has arrived at a final result for the scalar, given in the form . This final result can then be returned to the CPU. Consisting of a mere 4 integers, this is a trivially small amount of data to worry about any time of transfer here.

The benefit of this approach is that, having initialised (and sent to the GPU) all the parameterised subterm matrices of the original parameterised scalar, one can re-evaluate it for a whole new set of parameter values very efficiently. As all the parameterised data is still sat on the GPU, to re-evaluate the scalar one simply needs to send to the GPU a new set of parameter values (e.g. ) and rerun the kernels. Since, after initialisation, the only data that is being sent for each new evaluation is simply a handful of bytes to input the parameter values and a handful of bytes to return the final result, the data transfer time for new evaluations is negligible. (This is especially true if one sends the data for the next evaluation while the present one is still computing.)

In conclusion, while initialising the parameterised data and sending it to the GPU (it may be several gigabytes and hence take a non-trivial time to transfer among the hardware) may be a relatively slow task, compared to conventionally computing the scalar of a Clifford+T ZX-diagram, once it *is* initialised one can evaluate the scalar for new sets of parameter values very quickly. This is because most of the work has already been done in the initialisation and all that remains is to substitute in the parameter values and simplify the result – and, more to the point, this is done in a highly parallelised manner. The upshot is that taking some Clifford+T scalar ZX-diagram and evaluating it for various sets of input parameter values is exceptionally quick with this method (and with the more evaluations taken leading to an even better speedup as the evaluation runtime begins to render the slower initialisation runtime negligible by comparison).

*3.3. Application to repeated classical simulation*

Having now outlined a method by which a scalar Clifford+T quantum circuit may be reduced to its numerical scalar form, repeatedly for various sets of input values, it remains to showcase a useful application of this. One notable example is that of repeated classical simulation of a quantum circuit [This should be covered in the background section]. The idea here is to classically simulate a quantum circuit many times.

To classically simulate a circuit many times, using the methods above, the steps are much the same as those of [SECTION 2.2?], with a new, doubled (and then simplified) graph being generated for each successive output qubit. The difference here is that, in each step, instead of fixing the preceding outputs based on the probabilities found in the previous steps, the graph can be doubled and reduced for the general, parameterised case (leaving the outputs as ). Then, this parameterised graph, corresponding to one specific qubit output, can be evaluated any arbitrary number of times very efficiently. Each result here can be kept as part of its own bitstring to pass onto the next qubit’s graph.

In summary, for a circuit of qubits, this results in distinct, parameterised graphs, generated one at a time. For each in succession, it may be evaluated times (for repeat simulations) to find the simulated output of the next qubit, for each of the samples. Continuing this process for all qubits, allows the original circuit to be classically simulated times very efficiently – only ever needing to undergo the slow Clifford+T simplification routine of Clifford+T graphs, rather than .

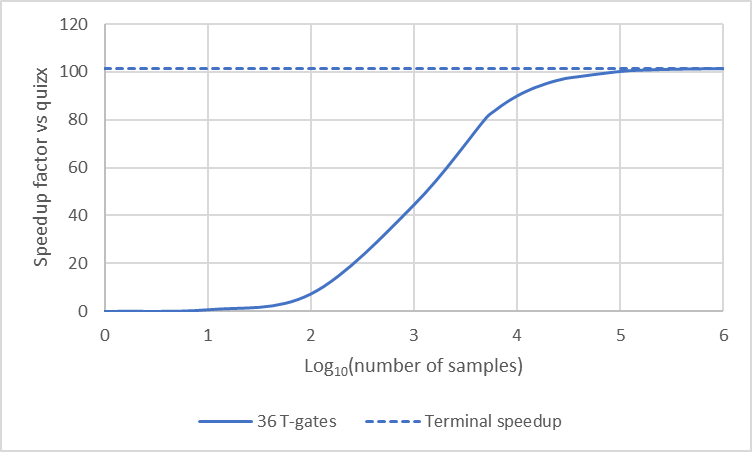
[Also discuss how it could be segmented with depth-first(?) If it became too much for gpu memory, etc.]

**4. Results**

With this method, the runtime of each successive sample, after initialisation, is expected to be considerably quicker than that of the conventional approach, as it does not require redoing any simplifications (just merely efficiently parallelised evaluations). However, the initialisation involved in preparing the parameterised scalar and transmitting the data to the GPU means that it will take a number of samples before the *overall* runtime is improved by this approach. In other words, if only a few samples are taken then the overall runtime will be dominated by that of *initialisation*, whereas as the number of samples increases, this initialisation runtime becomes negligible to the overall runtime.

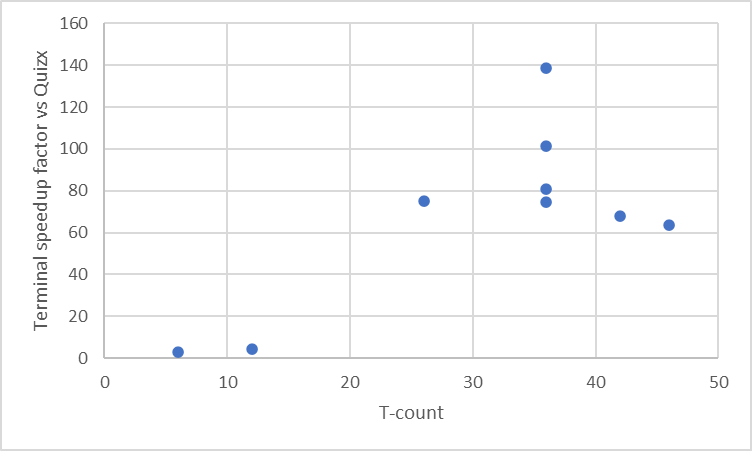
As such, one important metric by which to measure the efficacy of the method is the theoretical maximum (or *terminal*) speedup it can offer for a given circuit (that being the speedup achieved as the number of samples approaches infinity and hence the initialisation time becomes negligible). This is simply given by the ratio of the runtime of one repeat sample in this new method vs that of the old method. In conjunction with this, it will be worthwhile to analyse how many samples are required for a significant improvement in runtime to be attained.

For a given example of a 36 T-gate circuit of 20 qubits, of which 5 are parameterised, the following result was measured [show several examples here on the same plot? Also, specify the hardware specs]. Quizx took 472ms to sample this circuit, whereas the new approach, after an initialisation time of 5.944s, required just 4.656ms to compute a sample. This gives a terminal speedup of , meaning that, as the number of samples increases, this new method will run up to 101.4 times quicker than quizx for computing those many samples. Indeed, in this example, one need take only around 13 samples or more to see an improvement to the runtime. Better yet, as the graph shows, taking just hundreds of samples gives a significant speedup (of at least 44x), and taking thousands is sufficient to see speedups upwards of 90x.



[CAPTION, FIG 4.1]

The method outlined above was benchmarked against the more conventional, un-parameterised, approach of [REF KISSINGER/WETERING], for simplifying and taking many repeat evaluations of some random Clifford+T circuits. Such circuits of various depths, numbers of qubits, and T-counts were generated via PyZX based on [REF THE DOCUMENTATION FOR THE METHOD]. The terminal speedup factors (compared against Quizx) for these circuits are shown in figure 4.2 ahead, against their respective T-counts (after initial Clifford simplification).



[CAPTION, FIG 4.2]

[TODO – **Need a lot more data points for this plot of T-count vs terminal speedup, especially for higher T-counts (but these are very slow to run)**. Also, show for various numbers of qubits maybe??]

[DISCUSS THIS FIGURE]

As evidenced in this figure, for small T-counts this method offers no notable speedup versus Quizx. This is to be expected as, given small T-counts, the number of resulting terms is very small and hence is not able to take full advantage of the GPU parallelism (especially so if the number of terms is smaller than the number of parallel GPU threads).

**5. Conclusion and Future Directions**

[TODO]