npj | quantum information

Article

Published in partnership with The University of New South Wales

https://doi.org/10.1038/s41534-025-01133-7

Fault-tolerant quantum computation without distillation on a 2D device

Check for updates

Thomas R. Scruby[1], Kae Nemoto[1] & Zhenyu Cai[2,3]

We show how looped pipeline architectures—which use short-range shuttling of physical qubits to achieve bounded non-local connectivity—can efficiently implement the fault-tolerant non-Clifford gate between 2D surface codes described in (Sci. Adv. 6, eaay4929 (2020)). The shuttling schedule required is only marginally more complex than is required for implementing the standard 2D surface code in this architecture. We compare the resource cost with the cost of magic state distillation and find that, at present, this comparison is heavily in favour of distillation. The high cost of the non-Clifford gate is largely due to the relatively low performance of the just-in-time decoder used in the procedure, which necessitates very large code distances in order to achieve suitably low logical error rates. We argue that, as little attention has been given to the study and optimisation of these decoders, there are potentially significant improvements to be made in this area.

It is well known that all logical gates implementable by a constant-depth circuit in a two-dimensional topological code must belong to the Clifford group[1], and also that any quantum circuit composed entirely of Clifford operations is efficiently simulable on a classical computer[2]. Other techniques for performing logic in these codes (e.g. defect braiding) are similarly constrained[3], meaning that more complex and costly procedures must be employed in order to implement the non-Clifford gates necessary for a universal gate set. Typically these procedures involve the preparation of *magic states* which can be used to implement non-Clifford gates via a gate-teleportation circuit. The most commonly considered method of preparing magic states is distillation[4], which uses Clifford operations to transform many low-fidelity copies of a magic state into a small number of higher-fidelity copies. An alternative is to use a three-dimensional topological code where transversal non-Clifford gates can be implemented fault-tolerantly[5] and magic states can be prepared in constant time, but simulations of the performance of this approach suggest that it does not substantially improve space-time overhead[6] and it also comes with the added requirement of either a three-dimensional architecture or extensive non-local connectivity.

These issues can be avoided if we allow for logical operations implemented by circuits with non-constant depth. Making these operations fault-tolerant is more challenging as it can be difficult to correct for errors which occur at intermediate points in the circuit, but one example that can be made fault-tolerant is a linear-depth (in the code distance) CCZ gate between three 2D surface codes[7]. This gate implementation works by exchanging a spatial dimension for a temporal one so that the constant-time CCZ gate in the 3D surface code[8] becomes a linear-time gate in the 2D code. The fault-tolerance of this procedure comes from a decoding strategy called *just-in-time* (JIT) decoding[9] which attempts to guess the correction based on partial syndrome information of the corresponding 3D code at each time step of the circuit. The logical gate fidelity will therefore be lower than for the constant-time gate in the 3D code (where no guessing is required) but $O(d)$ fewer physical qubits are necessary for its implementation.

In addition to this non-standard decoding strategy, the linear time gate also requires a bounded amount of non-local connectivity, and so cannot be implemented on a device with strictly 2D connectivity. However, recent hardware developments have shown that this is a much less demanding requirement than it might once have appeared[10,11], and in particular the looped pipeline architecture in[12] uses short-range shuttling of physical qubits to provide exactly this kind of connectivity.

In this work we show exactly how to use a looped pipeline architecture to implement this gate and examine the corresponding resource cost relative to the cost of using the same architecture for magic state distillation. The physical operations required for this (short-range shuttling on fixed paths, single- and two-qubit physical gates, single qubit measurement in a Pauli basis) are identical to those required for the implementation of multiple copies of the standard planar surface code in this architecture, and the only differences in our case are the inclusion of local, constant-depth circuits that compile the physical CCZ gate and minor modifications to the shuttling sequence. Given that a fault-tolerance proof for the JIT decoder was presented in[7] we thus expect the entire protocol to be fault-tolerant.

Results

In this section we give a high-level overview of surface codes, the linear-time CCZ gate (Sec. II B) and the looped pipeline architecture (Sec. II C). Readers wishing to read about these topics in more detail can consult[7,13] and[12].

[1]Okinawa Institute of Science and Technology, Okinawa, Japan. [2]Quantum Motion, London, UK. [3]Department of Materials, University of Oxford, Oxford, UK.
e-mail: t.r.scruby@gmail.com; kae.nemoto@oist.jp; zhenyu.cai@eng.ox.ac.uk

Review of surface codes

The 2D surface code is typically defined via an assignment of qubits, X stabilisers and Z stabilisers to the edges, vertices and faces (respectively) of a square lattice[14], with the commutativity of the resulting stabiliser group ensured by the fact that there are always exactly two edges of a face sharing any given vertex. More generally, a surface code can be defined in the same way on any simple graph and commutativity will be guaranteed by the same property. This gives a natural extension of the code to other 2D and higher-dimensional tilings.

An alternate way to define a 2D surface code is to place qubits on the vertices of a 2D square lattice and assign X and Z stabilisers to alternate faces[15]. This approach also generalises to other 2D tilings provided the the faces can be two-coloured such that two faces with different colours either meet at an edge or not at all. Another example of such a tiling is the kagome lattice, where one type of stabiliser (X or Z) can be supported on hexagonal faces while the other type can be supported on triangular faces. Examples of 2D surface codes defined on square and kagome lattices can be seen in Fig. 6.

Logical qubits in surface codes can be created by defining the code either on a tessellation of a topologically non-trivial manifold or by defining appropriate boundaries. Specifically we can define two kinds of boundaries, which we will refer to as X and Z boundaries, and which correspond to places where strings of X/Z operators can terminate without anticommuting with any stabilisers. As can be seen in Fig. 6, a surface code encoding a single logical qubit typically has two X and two Z boundaries, with logical operators corresponding to strings of single-qubit X or Z operators connecting boundaries of the appropriate type.

As mentioned above, surface codes can also be defined on higher-dimensional lattices, and of particular relevance to this work is the 3D surface code. This code is most commonly defined on a cubic lattice in the fashion described previously[16], with logical Z operators still corresponding to strings of physical Z operators connecting pairs of Z boundaries, but logical X operators now taking the form of membranes of X operators stretching between four X boundaries. Additionally, in[8] it was shown that a set of three 3D surface codes can be defined using a 3D tiling of octahedra and cuboctahedra, with qubits on vertices and X and Z stabilisers on subsets of faces and cells. Remarkably, when defined in this fashion these three codes support a logical CCZ gate implemented by transversal application of physical CCZ between qubits of the three codes. This is the transversal logical operation that forms the basis for the procedure described in[7] which implements a linear-time logical CCZ between three 2D surface codes by trading a spatial dimension for a temporal one.

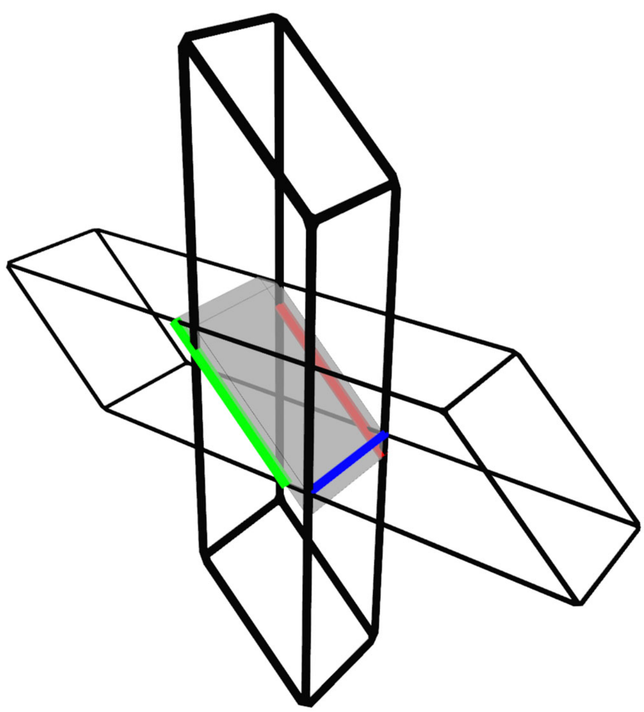


Fig. 1 | An example of three 3D surface code volumes which support a transversal CCZ. The thick black lines represent two overlapping codes while the third code is oriented in an orthogonal direction. When interpreted as a spacetime diagram for three 2D codes time for the two vertical codes runs bottom-to-top while time for the horizontal code runs left to right. A slice through the three codes is shown in grey and three logical Z operators are shown as coloured lines (note that although the red and green operators are parallel they run between different pairs of boundaries).

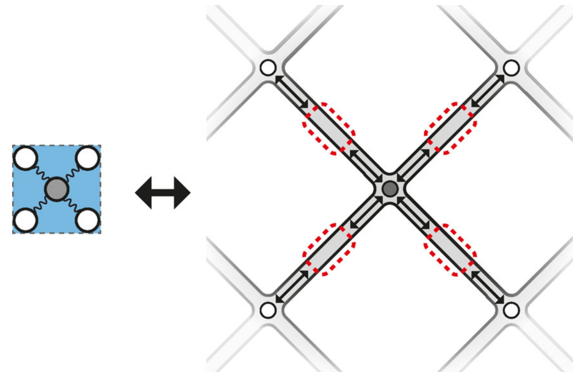
Surface code without distillation

Using the transversal (constant-time) CCZ available between three copies of the 3D surface code[8], we can prepares a CCZ magic state using the following steps:

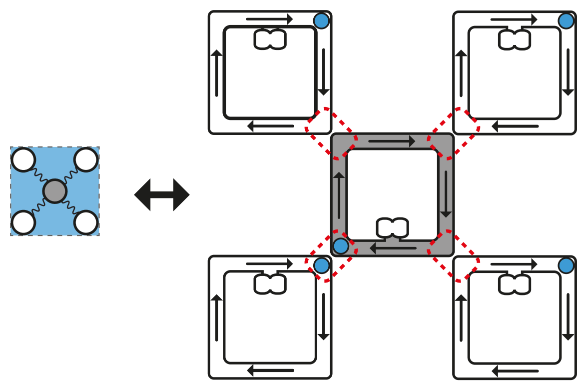
- Initialise all physical qubits in $|+\rangle$, measure all Z stabilisers and apply an X correction based on the outcomes of these measurements. This will prepare all three 3D codes in the logical $|+\rangle$ state.
- Apply transversal physical CCZ between the three 3D codes
- Measure out all of the physical qubits not supported on a particular set of boundaries to project the states of the 3D codes to states of 2D codes supported on these boundaries. Alternatively, the three 3D codes can be entangled with three 2D codes and then all the qubits of the 3D codes can be measured out to teleport their states to the 2D codes.

The linear-time CCZ is best understood as a reordering of the operations involved in the process above. In the linear-time gate, the sequence of operations experienced by each qubit is identical (barring differences in calculated corrections due to the different decoding strategies). The difference is that, for each of the three codes, the qubits are not all initialised at once. Instead, a thin slice of the 3D code is initialised, the physical CCZs are performed, and the physical qubits on the "bottom" of the slice are measured out while a new layer of qubits is initialised at the "top" (we will use the term "layer" to refer to a strictly 2D code while "slice" refers to a 3D code composed of a small number of layers). Individual CCZ gates are only applied to qubits after all Z stabilisers supported on that qubit have been measured, so the order of operations for each individual qubit is identical to the 3D case. After $O(d)$ timesteps, the spacetime history of the code will look like the full-distance 3D code and the final layer of qubits to be initialised will end up in the same state as the final 2D code in the constant-time procedure.

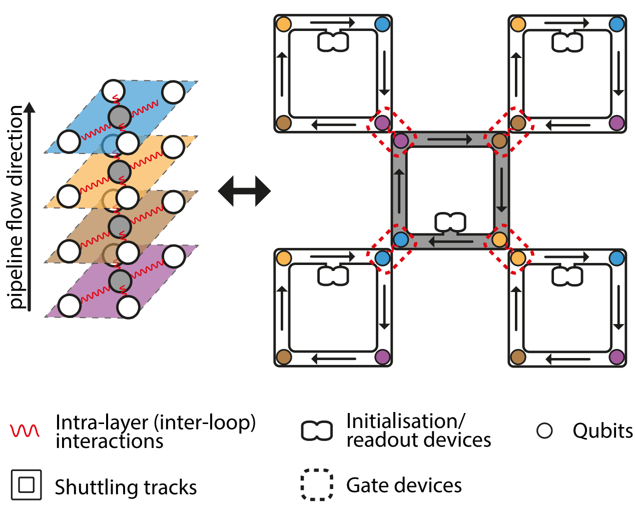
A practical implementation of this procedure would not be quite so simple. One notable issue comes from the fact that for the procedure to be fault-tolerant our slices through each code must contain distance d representations of the logical operators of that code. In each of these codes, the logical Z operators are stringlike, so we require there to be a Z logical operator representation lying in the plane normal to the time direction. However, for CCZ between three 3D surface codes to be transversal, the logical Z operators of the three codes must all be perpendicular and so we cannot choose a single time direction that preserves code distance in all three codes. Instead, at least one of the three codes must have a different time direction to the other two and our slices should be taken normal to the net time direction (see Fig. 1). We must additionally extend each of the three 3D codes along their respective time directions in order to preserve distance throughout the procedure and the result is that the three codes overlap on only a subset of qubits and CCZ is applied only between the qubits in this subset. Qubits not in this subset exist only to preserve the code distance and are not strictly necessary for the logical operation, meaning that the spacetime cost of the linear-time procedure is higher than for the constant-time version. Altogether this gives a procedure in which we begin with three patches of 2D surface code (two overlapping and one disjoint) which move towards, through each other, and then apart again as the gate is performed. The point of maximum overlap between the three patches is shown by the grey slice in Fig. 1.



(a) A five-qubit array in linear shuttling tracks.



(b) A five-qubit array in shuttling loops.



(c) Four five-qubit arrays in shuttling loops using pipelining .

Fig. 2 | Five-qubit arrays implemented using different shuttling architectures. Multiple qubits can be stored on the same loop through pipelining, which corresponds to multiple layers of five-qubit arrays. **a** A five-qubit array in linear shuttling tracks. **b** A five-qubit array in shuttling loops. **c** Four five-qubit arrays in shuttling loops using pipelining.

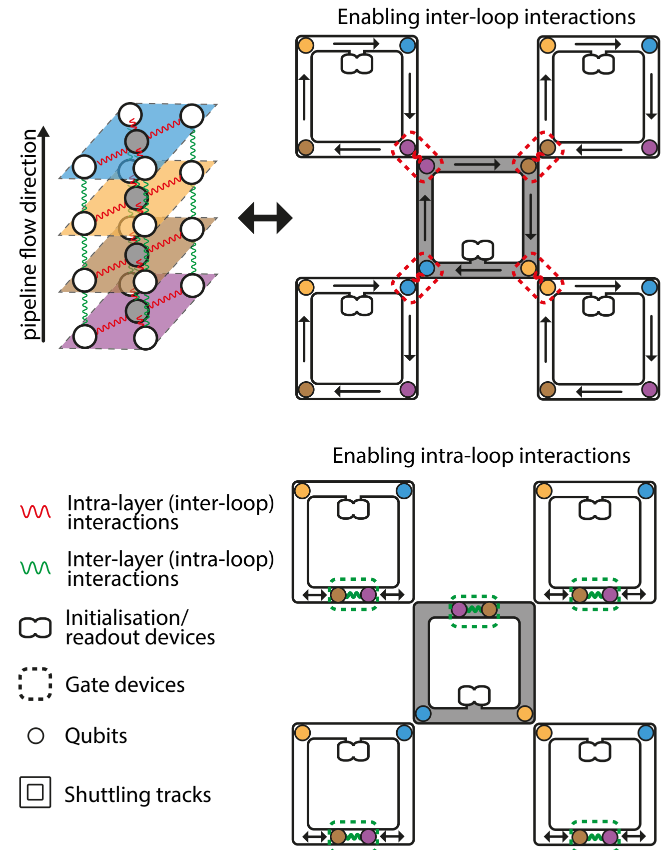


Fig. 3 | Interactions within layers are enabled by inter-loop (red) interactions while transversal interaction in between layers are enabled by intra-loop (green) interactions.

In Sec. IV we describe in detail how pipelining can allow for the implementation of a valid set of 3D slices in a strictly 2D device with local connections. For more details on the decoding strategies and fault-tolerance of the procedure readers should consult[7,13].

Looped pipeline

In many hardware architectures like semi-conductor spin qubits and trapped-ion qubits, in order to provide enough spaces for the control electronics and to avoid cross-talks, the qubits are often spaced out using linear shuttling highways and only brought together for two-qubits interactions[17–23] as shown in Fig. 2a. It was proposed in[12] that the above challenges also be solved by having qubits running around shuttling loops in synchronisation as shown in Fig. 2b. As the qubit at the centre loop moves around, it will meet and interact with qubits in the neighbouring loops one by one, carrying out all possible nearest-neighbour interactions after moving one full cycle around the loop. At first glance, such an architecture based on shuttling loops is no different from the linear version since they use the same amount of hardware. However, using the exact same looped architecture for one qubit array, we can fit additional qubits into each loop as shown in Fig. 2c. There the yellow qubits follow the exact same path as the blue qubits, just one step behind, thus forming a yellow-qubit array that is entirely independent of the blue array we had. In a similar way, as we fit K qubits in each loop, we can now store and process a stack of K qubit arrays in parallel with the exact same connectivity. Such a looped pipelined architecture thus allows us to increase the qubit density by K times without needing to add any additional hardware.

Now we have seen that interacting the corresponding qubits between different loops (inter-loop interactions) allows for intra-layer interactions within the stack of qubit arrays. If we interact with the qubits within the same loops (intra-loop interactions) as shown in Fig. 3, then we are effectively interacting with the corresponding qubits in between the different layers of qubits, i.e. performing transversal operations between qubits. This effectively gives us a limited amount of 3D qubit connectivity on a 2D platform through pipelining.

One needs to note that even though shuttling might be useful for arbitrarily increasing qubit connectivity in small experiments, it remains a challenge to devise efficient shuttling schemes in a scalable way taking into account the shuttled distance, scheduling, etc. The looped pipeline

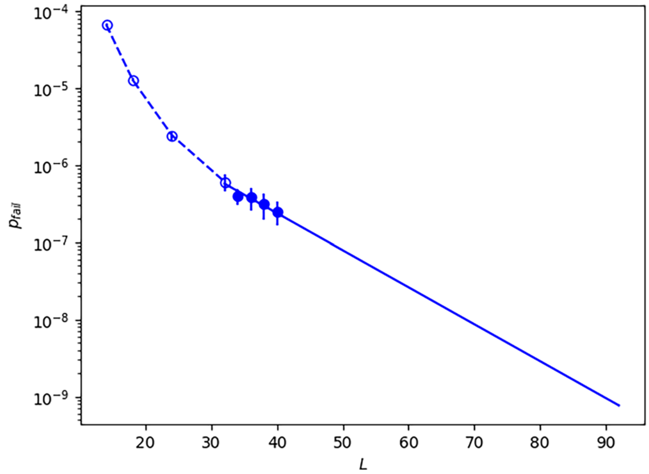


Fig. 4 | Logical X error probability (p_{fail}) in a single surface code (the red code described above) vs distance (L) when performing a linear-time CCZ under phenomenological noise, using code from[13,29] and physical error probability $p = 4.67 \times 10^{-4}$. Between 1×10^8 and 2×10^8 samples were taken for each value of L. Empty points are data from[13] while filled points are new data produced for this work. The solid line is a linear fit to the five data points in the range $30 < L \le 40$. We emphasise that this performance is dependent on the JIT decoder and not an inherent property of the linear-time CCZ, and so overheads would improve given an improved decoder. Data for this plot is available at[40].

architecture is such a *scalable* scheme for increasing qubit connectivity that is applicable to platforms like silicon spins, trapped ions and neutral atoms.

Problem setting

In this section, we will compare the resource overhead of the architecture with and without magic state factories. We will consider the case of implementing parallel CCZ gates covering all qubits, which is a bottleneck step in the conventional approach with magic state factories if we want to implement the carry lookahead adder in[24] (to be exact, it does not require parallel CCZ covering exactly all qubits, but it is a good approximation). Adders are key components in Shor's algorithm, and to factor 2048 bit numbers, we need around $N = 6000$ logical qubits, $D_{\mathrm{cycle}} = 2.5 \times 10^{10}$ code cycles and $M_{\mathrm{CCZ}} = 3 \times 10^9$ CCZ gates[25]. Hence, our targeted logical error rate per code cycle and our target logical CCZ error rate are $P_{\mathrm{cycle}} = 1/N/D_{\mathrm{cycle}}$ $\sim 6 \times 10^{-15}$ and $P_{\mathrm{CCZ}} = 1/M_{\mathrm{CCZ}} \sim 3 \times 10^{-10}$, respectively. Note that here we are assuming magic state distillation is not the bottleneck when we are calculating P_{cycle}.

We will be considering the case where the physical gate error is $p = 5 \times 10^{-4}$. Using the formula $P_{\mathrm{cycle}} = 0.1(100p)^{(d+1)/2}$ [26], the minimum code distance d we need to achieve the target logical error rate $P_{\mathrm{cycle}} \le 6 \times 10^{-15}$ is $d = 21$.

Using the CCZ factories outlined in ref. 27, fixing the level-2 distance at $d = 21$, the level-1 distance d_1 needed to achieve the target logical error rate $P_{\mathrm{CCZ}} \le 3 \times 10^{-10}$ is $d_1 = 13$ (outputs $P_{\mathrm{CCZ}} \sim 10^{-12}$). The footprint of the CCZ factories is shown in Fig. 1 of ref. 27, which is $12d_1 \times (16d_1 + 4d) \approx 7.5d \times 14d$, for our case. The magic state production rate is limited by the level-1 distance in this case since $d_1 {*} 2 + 1 \ge d$. Hence, the rounds of repetition needed for operations like lattice surgery is $d_1 \times 2 + 1 = 27$, and we need 8.5 rounds, which means the number of code cycles needed to produce one CCZ state is $27 \times 8.5 \approx 230$. If we allow consecutive rounds of magic state production to overlap, we can potentially reduce the number of rounds needed from 8.5 to 5.5[27], but we will not make use of this here for simplicity. We need to implement one round of logical CNOT for T gate teleportation, which will take $2d$ code cycles using lattice surgeries. After the teleportation, we require up to 3 (and on average 1.5) CZ corrections based on the teleportation measurement result, which adds in $1.5 \times 2d = 3d$ more code cycles. Hence, the total number of code cycles needed to implement a CCZ gate from the beginning of magic state distillation is $T_{\mathrm{msd}} = 230 + 2d + 3d$ ≈ 330. Our discussion so far is entirely about 2D surface code architecture without looped pipelines, and thus no transversal entangling gates are used. Naive implementation of looped pipelines will enable a stack of such CCZ factory to be implemented in parallel, which we will compare against linear CCZ in the next section. It will also be interesting to further optimise the whole CCZ factories using transversal operations in future work[28].

Comparison to linear in-place CCZ

To estimate the code distance d_{CCZ} required to achieve a logical error rate $P_{\mathrm{CCZ}} \sim 3 \times 10^{-10}$ with the linear time CCZ, we can extrapolate from numerical data. The only prior data available is from ref. 13. The decoder used in that work can be understood at a high level as a modified version of minimum-weight perfect matching in which repeated occurrences of the same syndrome result in alterations of the edge weights in the matching graph. However, due to finite-size effects, meaningful extrapolation cannot be performed using only the distances simulated in that work. Therefore, we use the same code[29] to perform additional simulations at larger distances as shown in Fig. 4. We note that detailed fitting (e.g. to estimate a threshold and assess sub-threshold scaling) was already performed in ref. 13, and the same analysis applies here. Extrapolating from these data, we estimate the required distance to be $d_{\mathrm{CCZ}} \sim 100$. We emphasise that this estimate is unreliable for a number of reasons, such as the large amount of extrapolation employed and the lack of a circuit-level noise model in these simulations, and thus the real-world performance can be worse than this. Additionally, the slices simulated in ref. 29 are not the same slices we have described above and so some amount of variation in performance can be expected to result from this. Regardless, we consider $d_{\mathrm{CCZ}} \sim 100$ to be a suitable lower bound on the code distance required and note that larger values of d_{CCZ} would not change any of the conclusions of the following analysis.

Given $d_{\mathrm{CCZ}} \sim 100$, the footprint of each logical qubit during the linear CCZ gates is expanded into $d_{\mathrm{CCZ}} \times 2d_{\mathrm{CCZ}} \approx 5d \times 10d$. A natural architecture that allows the implementation of linear CCZ is shown in Fig. 5b, where we have these $5d \times 10d$ patches laying side-by-side. Each patch in Fig. 5b actually represents a stack of such patches using the looped pipeline architecture, CCZ gates can be implemented on almost all logical qubits in parallel (except for those at the boundary) by simultaneously sliding the layers in all patches. The same layer in different stacks will represent the same code out of the three and thus will slide in the same direction, while different layers in the same stack may slide in different directions since they can represent different code configurations out of the three. When not implementing the linear CCZ, the surface code patches will shrink to the size of $d \times d$ as shown in Fig. 5a, where we have a corridor with width $4d$ in one direction and $9d$ in the other.

As shown in Fig. 5c, this is actually enough to fit in one CCZ factory per 3 logical code patches, and in the looped pipeline architecture, this translates into a stack of CCZ factories per 3 stacks of logical code patches. If we want to perform CCZ gate over all qubits in this layout, we only need to output one magic state per magic state factory plus the time needed for gate teleportation, which is $T_{\mathrm{msd}} \approx 330$ as derived above. In Sec. IV C, we have calculated the time overhead of the linear CCZ gates to be $T_{\mathrm{lin}} = 6d_{\mathrm{CCZ}} = 600$. We also need to add in the d_{CCZ} rounds needed for expanding the code patch from d to d_{CCZ}, thus the total time for the in-place CCZ is $T'_{\mathrm{lin}} = 7d_{\mathrm{CCZ}} = 700$. Hence, the space needed for implementing linear CCZ is enough for fitting a CCZ factory in place right next to the target qubits, and the time required for the CCZ factory to produce and teleport a CCZ gate is around $T'_{\mathrm{lin}}/T_{\mathrm{msd}} \approx 2$ times faster than linear CCZ.

Comparison to linear CCZ factories

The 2-times speed up here is for the case in which we want to apply parallel CCZ to all of the qubits. In practice, in most time steps we will not require so many CCZ gates such that one factory per three logical qubits is a gross overestimation. In fact, 14 factories are enough for magic state distillation to be a non-rate-limiting step in our 6000-qubit problem of factoring a 2048-bit integer using ripple carry adders (28 factories if using carry runways adders)[25]. In these regimes, the architecture in Fig. 5 that allows in-place

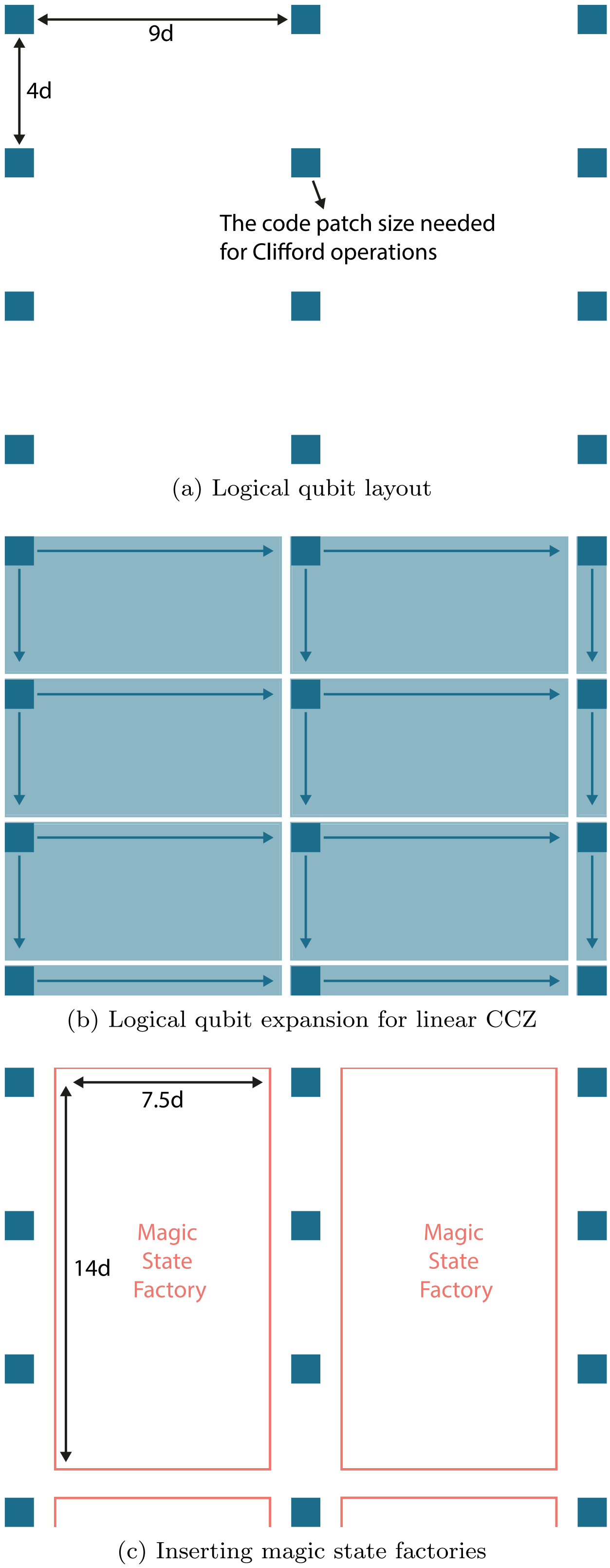


Fig. 5 | Each patch here actually represents a stack of *K* code layers where *K* is the number of qubits per loop in the looped pipeline architecture. Similarly the magic state factory here actually contains a stack of *K* magic state factories. **a** Logical qubit layout. **b** Logical qubit expansion for linear CCZ. **c** Inserting magic state factories.

CCZ gates everywhere will lead to a massive waste of space. Instead, one may want to create space only at the logical qubits and at the time where we want to implement the CCZ gates, by moving other logical qubits around.

Alternatively, we can also use linear CCZ to implement CCZ factories. We can construct such factories using a row in Fig. 5b, a row of M stacks with 3 layers per stack (and 6 layers during CCZ implementation) will give us $M - 1$ linear CCZ factories. Hence, the average space overhead of a $\sim d_{\mathrm{CCZ}} \times 2Md_{\mathrm{CCZ}}/(M-1) \approx 50d^2$-size stack per linear CCZ factory (for $M \gg 1$). In this model of fitting three layers per stack, three distillation factories can be fit into a stack of the size $7.5d \times 14d$ as derived above, i.e. the space overhead is on average a $7.5d \times 14d/3 = 35d^2$-size stack per CCZ distillation factory, this is 1.4 times smaller than the linear CCZ factory. Compared to the in-place CCZ gates, we do not need to expand the code patch from d to d_{CCZ} in the linear CCZ factory, thus the time overhead is reduced from $T'_{\mathrm{lin}} = 7d_{\mathrm{CCZ}}$ to $T_{\mathrm{lin}} = 6d_{\mathrm{CCZ}}$. However, such a reduction is marginal, and thus the linear CCZ factory is still around 2-times slower than the CCZ distillation factory. The routing overhead, etc, will be similar for both the linear CCZ and the CCZ distillation factories. In this setting, it is also possible to achieve a space-time tradeoff by using a standard 3DSC X error decoder to check the correction produced by the JIT decoder after the gate is complete, and then post-select on outcomes where these two decoders agree. However, we do not expect that this can be used to make the linear CCZ competitive with distillation as the latter already outperforms the former in both space and time cost, and Fig. 4 suggests that even a small reduction in code distance would lead to a significant reduction in the success probability.

In our consideration here, we have made a range of favourable assumptions for the linear CCZ gates. We assume the expansion of layers into slices in the implementation of the linear CCZ does not exceed the number of qubits allowed per loop and does not increase the code cycle time. Furthermore, the distance 100 that we are assuming here for linear CCZ is not based on a circuit-level error threshold, thus the actual distance needed for linear CCZ can be even higher. The CCZ factories we use here are simply the ones designed for a 2D layout, and we can fit multiple of them in a stack using the looped pipeline architecture. There is an additional possibility of taking advantage of the transversal entangling operations within the stack to further reduce the space overhead of the CCZ factories[12], which we have not explored here.

Discussion

We have shown that the linear-time CCZ gate of ref. 7 is highly compatible with looped pipeline architectures, requiring a qubit shuttling schedule that is only slightly more complex than the standard surface code. This provides us with a valuable example of the utility of this architecture for the practical implementation of fault-tolerant CCZ, demonstrating how complex codes and fault-tolerant protocols can be straightforwardly implemented using a fairly minimal set of almost local operations. On a less positive note, the practical spacetime overheads for this particular fault-tolerant protocol still exceed those of magic-state distillation, even under a range of favourable assumptions. More specifically, if we want to perform in-place linear CCZ of error rate $P_{\mathrm{CCZ}} = 10^{-10}$ among all logical qubits, the distance needed for linear CCZ will create a corridor large enough to fit a CCZ distillation factory next to every triple of logical qubits as shown in Fig. 5, so there is no advantage in the space overhead. The speed of CCZ distillation is two times faster than linear CCZ in our estimate.

The large overhead of the linear-time CCZ is mostly due to the low threshold and poor sub-threshold scaling of the JIT decoder. Hence, we expect the performance gap between these approaches to decrease when we have a smaller target CCZ error rate. For example, if we are targeting $P_{\mathrm{CCZ}} = 10^{-7}$ instead, we will require $d_1 = 9$ and $d = 15$ for magic state distillation, which translates into a space overhead of $7d \times 14d$ and a time overhead of $(d_1 \times 2 + 1) \times 8.5 + 5d = 236.5$ code cycles. For linear CCZ, we will need $d_{\mathrm{CCZ}} = 50$, which translate into $3.3d \times 6.6d$

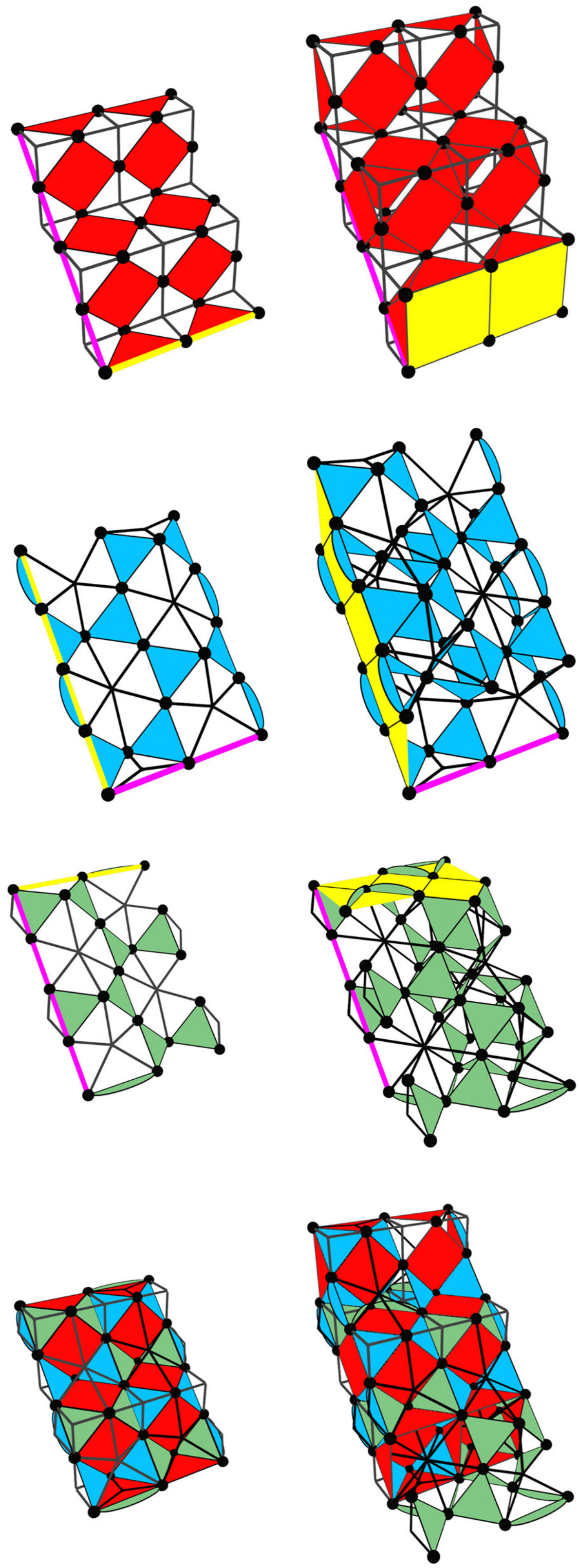


Fig. 6 | Layers and slices through each of the three codes + their overlap. Black circles are qubits, black stars are X stabilisers and coloured faces are Z stabilisers (different colours are only used to differentiate the three codes). Logical Z representations are shown by pink lines and logical X representations are shown by yellow lines/membranes.

and $7d_{\mathrm{CCZ}} = 350$ code cycles for space and time overhead, respectively. In this case, the space in the corridor created for implementing linear CCZ is no longer enough to fit one CCZ factory per 3 logical qubits, thus we need to consider how CCZ resource states are stored, shared and routed when using CCZ factories. The time improvement is also reduced from 2 times to 1.5 times. The CCZ distillation factories approach will still have the advantage taking into full consideration the various additional factors mentioned at the end of the previous section, but such an advantage is reduced compared to when $P_{\mathrm{CCZ}} = 10^{-10}$.

One also needs to keep in mind that magic state distillation, being a method that uses post-selection, an exponential suppression in the output error rate will require an exponential increase in the number of trials (i.e. an exponential decrease in the pass rate of the checks). This characteristic is even more prominent in the recent related cultivation scheme[30], which only targets a specific range of output error rates. On the other hand, the linear CCZ being a fault-tolerant approach only requires a linear increase in the code distance to achieve exponentially stronger error suppression, even though such an advantage over magic state distillation will only occur at ultra-small target CCZ error rate far below the regime of practical interests with the current schemes. Thus more interestingly, one can look into ways to merge the philosophy of post-selection from the distillation/cultivation schemes into the linear CCZ schemes in order to achieve a better balance between asymptotically good behaviour and practical performance. The decoder cross-checking method mentioned previously is one such idea.

It is also worth noting that the JIT decoder used to obtain these performance estimates is (to our knowledge) the only such decoder that has been investigated numerically, and there is no reason to believe that it is optimal, or even close to optimal. Improved decoders that have higher thresholds or better sub-threshold scaling would also lead to significant reductions in the associated resource costs for the linear-time CCZ. Recently, more sophisticated versions of the protocol studied in this work have been proposed[31,32] and these may provide additional perfomance improvements independent of those arising from improved JIT decoders, or may allow for new opportunities to improve these decoders. Even if these improvements by themselves cannot make the linear-time gate competitive with magic state distillation in the practical regime, especially with the continuous progress in magic state distillation[30], when combined with other factors (e.g. lower target logical error rates as discussed above) there may exist regimes where these approaches have a similar cost.

Methods

In this section, we describe how to use the looped pipeline architecture described previously to implement the error correcting codes and linear-time non-Clifford gate described in[7].

Code details

The slices of 3D code used to perform the linear-time CCZ cannot be chosen arbitrarily and only certain choices will preserve the code distance while also being compatible with the CCZ gate and JIT decoder. Two proposals for valid slices exist in the literature[7,13]. The first (one that is consistent with Fig. 1) has the advantage of being composed of only two layers per slice, whereas the second requires three layers of qubits per slice but allows for a qubit initialisation and measurement sequence that is synchronised across the three codes. In this work, we will use the former set of slices as fewer layers lead to fewer qubits per loop, which should simplify scheduling and reduce errors.

Examples of distance-3 single layers and two-layer slices for the three codes are shown in Fig. 6, which have different lattice structures[8]. In the first of the three (red faces) the 2D layers are equivalent to surface codes defined on a square grid with qubits on vertices and X and Z stabilisers on alternating faces. The 2D layers in the other two codes (blue and green faces) are defined

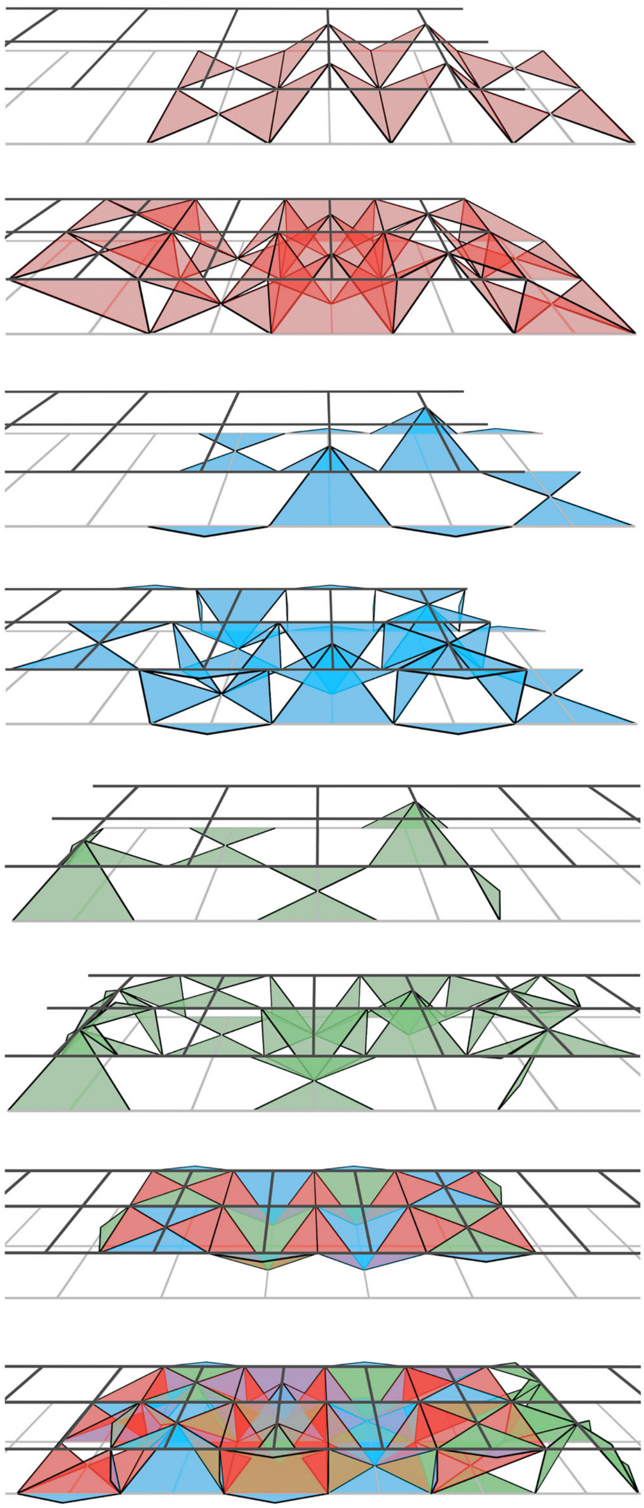


Fig. 7 | Mapping of the layers and slices from Fig. 6 to a pair of square grids with qubits on edges. Only Z stabilisers are shown. The assignment of the qubits of the first layer to the top or bottom grid is arbitrary, but the choice shown here gives a nice symmetry between this and subsequent layers.

with qubits on the vertices of a kagome lattice, Z stabilisers on triangular faces and X stabilisers on hexagonal faces. These two codes are mirrored with respect to each other.

We assign a vertical time direction to the red and blue codes and a horizontal time direction to the green code. In each case the 3D slice is obtained by taking a copy of the layer and translating it one lattice unit in the relevant time direction (and enforcing any appropriate intermediate stabilisers from the 3D code bulk). For the blue and green codes we must also mirror the translated layer along one axis and add extra qubits which lie between the top and bottom layers but are not part of either layer. Boundary stabilisers for each of the three codes are chosen so that logical Z operators run between a different pair of boundaries in each case (more obvious in the slices than in the layers). The final two images of Fig. 6 show the layers and slices at their point of maximal overlap (corresponding to the shaded slice position shown in Fig. 1).

Pipelined implementation

Before describing an implementation of these slices using a pipelined architecture it is helpful to first consider a mapping of to a pair of square grids with qubits on edges as shown in Fig. 7. The first red and blue layers are initialised at the right-hand end of the two grids and subsequent layers are shifted to the left relative to these first layers. The first layer of the green code is initialised at the left-hand end of the grids and subsequent layers will be shifted to the right. This motion corresponds to the projection of the layers and slices onto the plane normal to the sum of both time direction vectors in Fig. 1.

To obtain a pipelined implementation of these layers and slices, we first combine the two grids from Fig. 7 into a single grid (by associating edges which differ only in their z coordinate). Then we assign one loop to each vertex, edge and face of this grid. Each of these loops will contain 0, 1 or 2 qubits per code as required. A detailed description of this implementation and the shuttling schedule for each code will be discussed below.

A useful feature of the linear-time CCZ procedure is that we only need to measure the Z stabilisers of the slices and not the X stabilisers (only the Z stabiliser measurements are required to initialise the code and errors will not accumulate on the physical qubits because they are being constantly measured out and reinitialised). In fact, we could not measure the X stabilisers even if we wished to because they do not commute with the CCZ gate, so they can be measured only in the first and last layers (at the start and end of the procedure) and not at any point in between. Note that this does not mean that the single-qubit X measurements themselves are not used, or that we can completely disregard the effect of Z errors occuring during the protocol. Instead, these measurement outcomes are used to compile an X stabiliser syndrome for the full 3D code in postprocessing and this syndrome is used (in conjuction with 2D stabiliser measurements at the end of the protocol) to compute a Z correction for the final 2D code. This is convenient because the X stabilisers have fairly high weights and become highly non-planar when mapped to 2D. The Z stabilisers, on the other hand, are at most weight-4 and can be mapped to 2D much more easily. Accordingly, our pipelined implementation will allow for measurement of X and Z stabilisers in the layers but only for measurement of Z stabilisers in the slices.

Figure 8 shows the pipelined implementation for each of the three layers. The first layer is simply the familiar 2D surface code on a square lattice and the implementation we obtain for this code is identical to the one described in[12]. The second and third images in Fig. 8 show implementations of the two kagome lattice layers. The most natural implementation these layers would measure each hexagonal X stabiliser using a single large ancilla loop, the qubits of which could interact with all six data qubits in the stabiliser's support. However, we wish to use the same set of loops for all three codes so instead we measure these stabilisers using a two-step process. For the blue code (and in the bulk of the green code) we have two types of ancilla loop for each X stabiliser, which we label **A** and **B** (notice that the **B** loops are ancilla qubit loops in these codes but data qubit loops in the red code). To measure the stabiliser the qubits in the two **A** loops are first entangled with all three adjacent data qubits, then they are both entangled with the qubit of the **B** loop which can then be measured out to complete the stabiliser

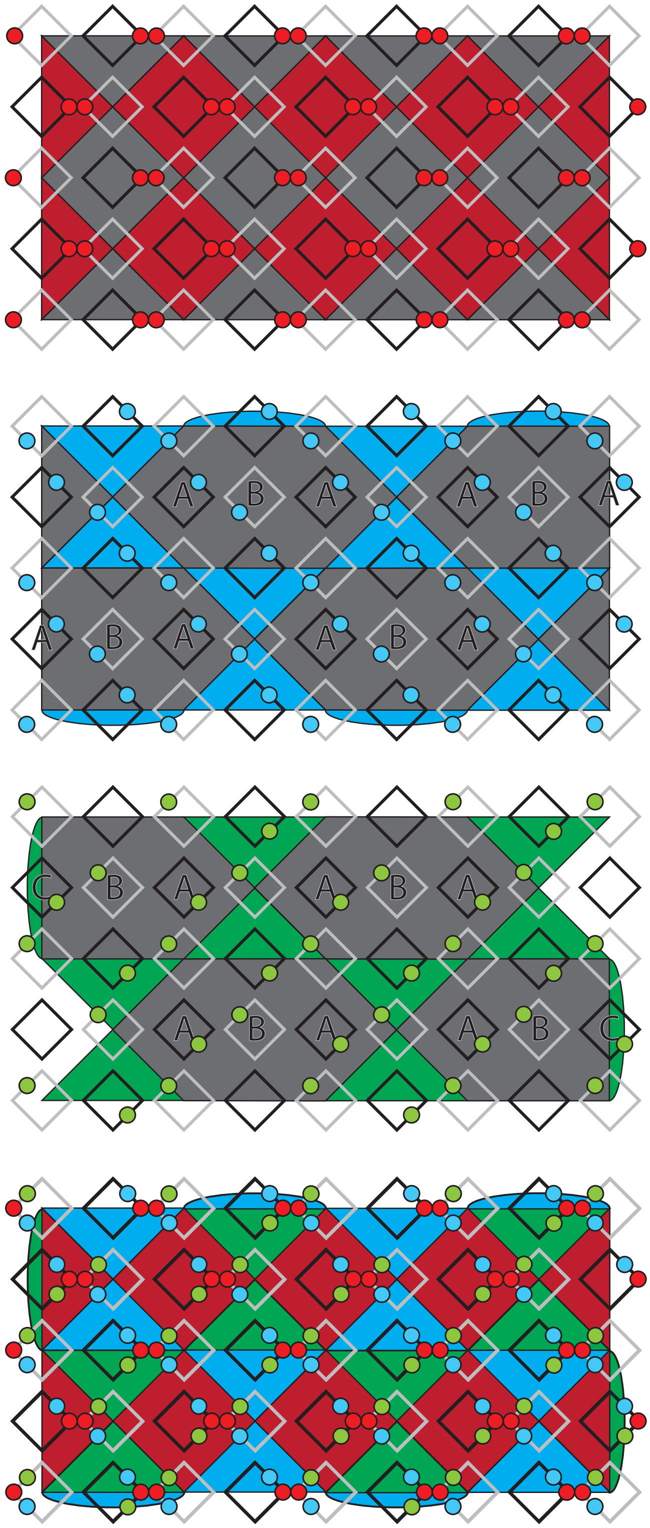
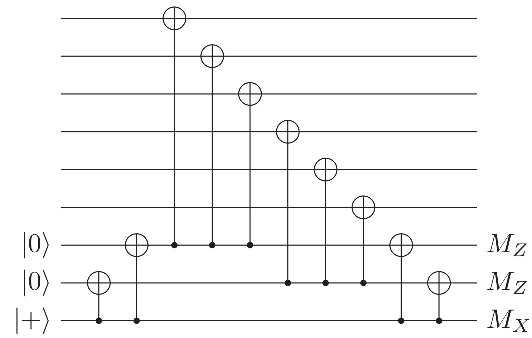


Fig. 8 | Pipelined implementation of the three 2D layers from Figs. 6, 7. Coloured (grey) faces are Z (X) stabilisers. Grey loops contain data qubits and black loops contain ancilla qubits. Qubits move clockwise in all loops. The final image shows the overlap of all three layers. Only Z stabilisers are shown in this case as this overlap only occurs at the midpoint of the procedure when we do not need to measure X stabilisers.

measurement. An example of a suitable circuit is



where the two qubits prepared in $|0\rangle$ are the ancilla qubits in the **A** loops and also function as flag qubits, while the qubit prepared in $|+\rangle$ is the ancilla in the **B** loop.

At the Z boundaries of the green code we have an additional loop type (**C**) whose ancilla qubit is responsible for measuring both a Z boundary stabiliser and two of the qubits of an X stabiliser. This is not a problem provided we first measure all Z stabilisers and then all X stabilisers rather than all in parallel (this is actually required even for the red code[12]).

As mentioned above, we do not need to directly measure the X stabilisers of the slices so our proposed implementation (shown in Fig. 9) only allows for Z stabiliser measurements. Unlike in the layers, some of the loops now contain two qubits (one from each layer in the slice). In this setting we can consider three different kinds of qubit, those which belong to only a single layer, those which belong to both layers, and those which belong to the slice but to neither layer. We also have two different kinds of Z stabiliser which we refer to as intra-layer and inter-layer stabilisers. Intra-layer stabilisers are those which are supported entirely on qubits belonging to only a single common layer, while inter-layer stabilisers are all other stabilisers. Note that even stabilisers fully supported within a single layer are still regarded as inter-layer stabilisers if some qubits in their support belong to both layers. All data-ancilla interactions necessary for measuring the intra-layer stabilisers can be performed in a single loop cycle. However, the ancilla qubits for the inter-layer stabilisers are in phase with the data qubits from only one of the layers, but in antiphase with the data qubits of the other layer. We therefore propose the following three-step procedure for measuring all the stabilisers of these slices

1. Perform one full loop cycle. All interactions for intra-layer stabilisers and all possible interactions for inter-layer stabilisers are performed.
2. Advance all inter-layer ancilla by half a cycle.
3. Perform one more full loop cycle. The remaining operations for the inter-layer stabilisers can be performed in this cycle.

The operations performed in step 1 are similar to that of a code cycle in the 2D codes. Step 2 consists of pure shuttling, which is usually not the rate-limiting step in many hardware[21,33]. The number of operations performed in step 3 is fewer than the usual code cycles since we are only interacting with the data qubits in one of the layers for the inter-layer stabilisers. Hence, we will approximate the total time needed in steps 2 and 3 as one code cycle, which means the full process of the three steps above takes around two code cycles.

Because the partitioning of loops into "data" and "ancilla" is consistent across all three slices (unlike with the layers) the half-cycle advancement of the ancilla qubits does not interfere with the scheduling of any other qubits. Even in the case where intra- and inter-layer ancillas share a loop there are no issues because the intra-layer ancillas have already completed all necessary interactions.

For the first image in Fig. 9 the correspondence between inter-layer stabilisers and loops is straightforward as each loop containing one (two)

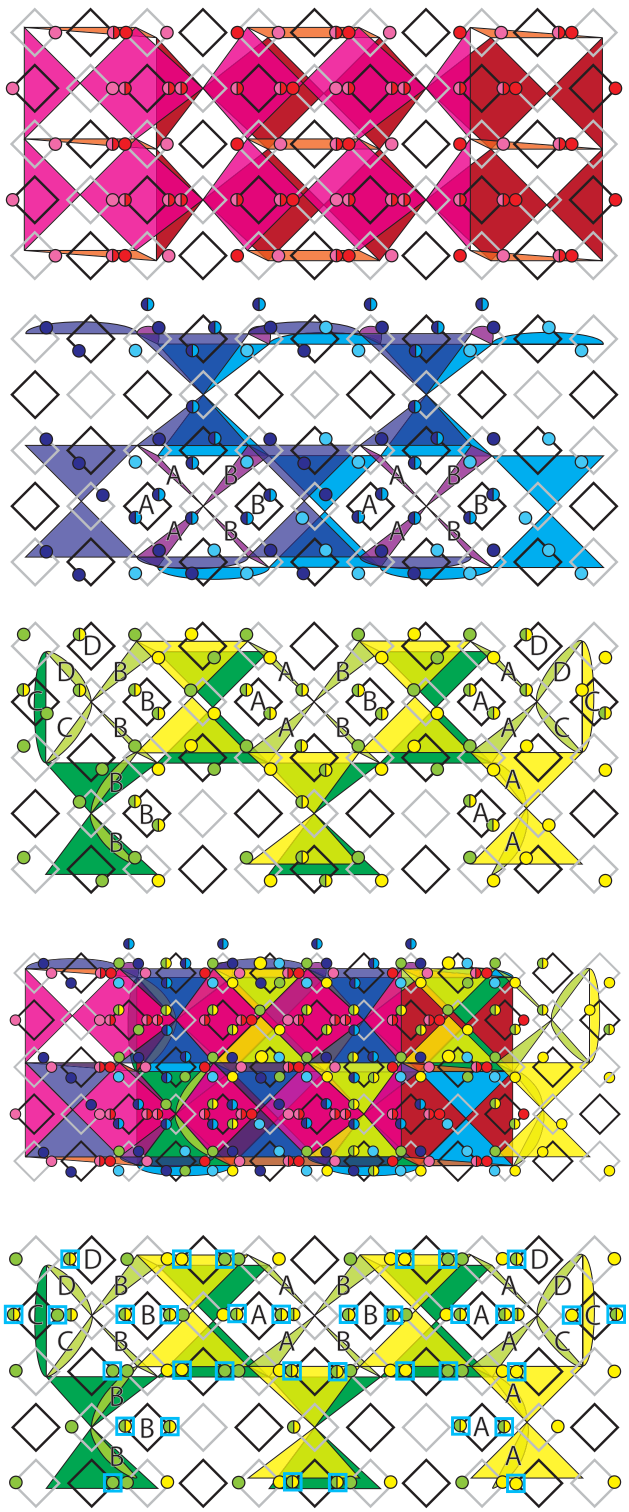


Fig. 9 | Pipelined implementation of the three slices from Fig. 6 and 7. The three colours of faces in each slice distinguish Z checks that exist in the bottom layer, top layer, and only in the slice. Single-coloured (two-coloured) circles are data qubits belonging to only one layer (neither/both layers) or ancilla qubits for intra-layer (inter-layer) stabilisers.

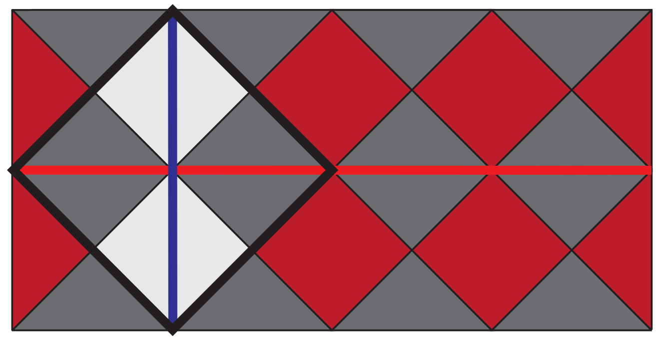


Fig. 10 | Conversion of a distance-3 rotated surface code (thick black lines) to the red code shown previously. New Z stabilisers are shown in red. X (blue) and Z (red) logical operator representations are also shown.

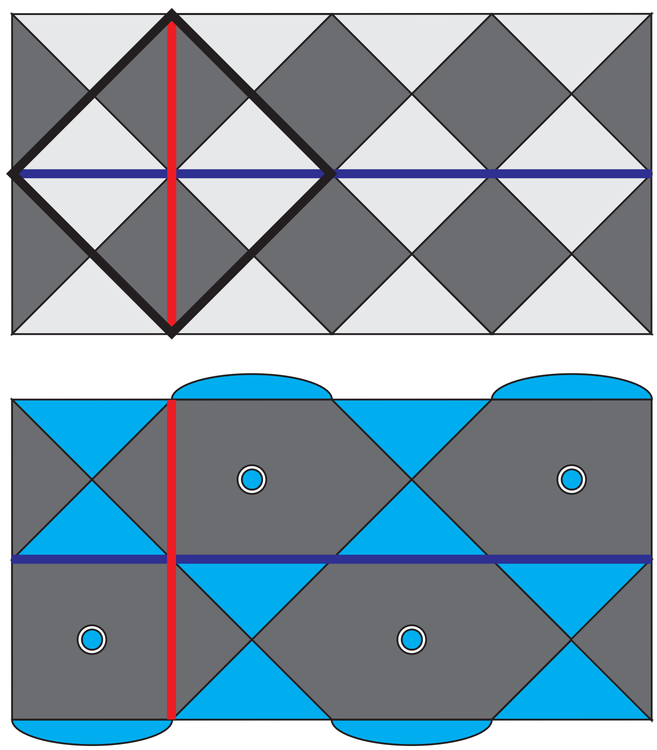


Fig. 11 | Conversion of a distance-3 rotated surface code to the blue code shown previously. The code is first expanded to an unrotated code and then some qubits are measured out in the Z basis (blue circles). X checks in the new code are products of checks from the old code, while Z checks are checks from the old code restricted to their supports on unmeasured qubits. The displayed logical operator implementations are valid for both initial and final codes.

inter-layer ancilla overlaps with exactly one (two) of the inter-layer stabilisers. This is less straightforward in the second and third images and additional modifications are required to accommodate the boundary stabilisers in these codes. In Fig. 9 the inter-layer ancillas in the loops labelled **A** measure the two inter-layer stabilisers (also labelled **A**) to their right while those labelled **B** measure the two inter-layer stabilisers to their left. Additionally, the green code has loops labelled **C** and **D** which each contain one inter-layer ancilla that measures the adjacent stabiliser with that same label. The **C** loops also contain an intra-layer ancilla that measures the intra-layer stabiliser which overlaps with the loop. Finally, one boundary of the blue code supports boundary stabilisers which are supported on two qubits of the same loop. In isolation we could measure these stabilisers by interacting these two qubits with each other or by adding an ancilla qubit to this loop, but when multiple slices overlap these two qubits will be separated by qubits from the other codes. Instead, we introduce static ancilla qubits next to this boundary which can interact with the data qubits as they move past.

Step-by-step examples of the shuttling and interaction schedules for each of the three codes are provided in the supplementary information.

In the last image we show the overlap of all three slices. Notice that in the loops which contain six qubits these qubits are arranged in two groups of three so that the same-layer qubits from all three codes are together. These are the triples of qubits between which we need to apply CCZ, so this arrangement makes the application of this gate more straightforward.

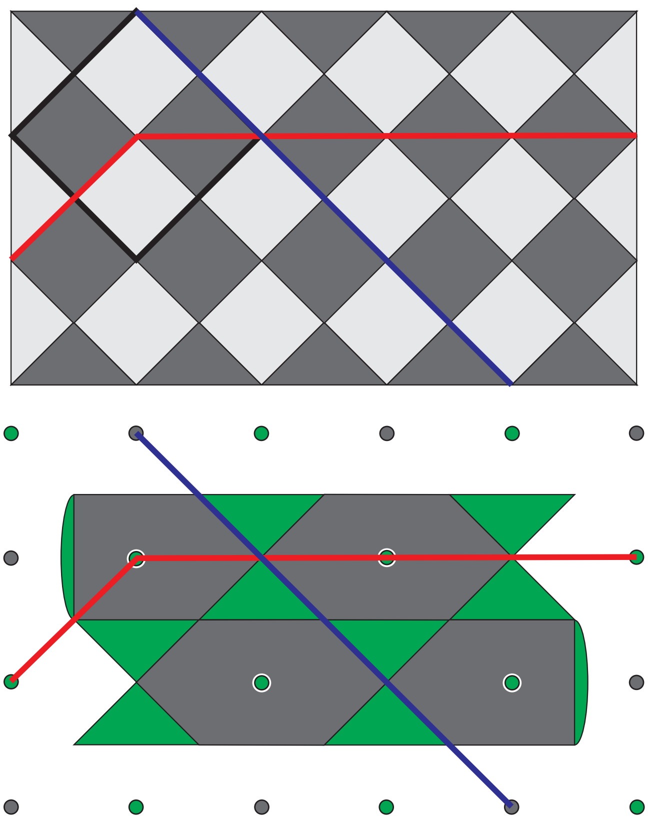


Fig. 12 | Conversion of a distance-3 rotated surface code to the green code shown previously. The code is first expanded to a large unrotated code and then some qubits are measured out in either the X or Z basis (green for Z and grey for X). X and Z checks in the new code are (products of) checks in the old code restricted to their supports on unmeasured qubits. The displayed logical operator implementations commute with all new stabilisers and single-qubit measurements.

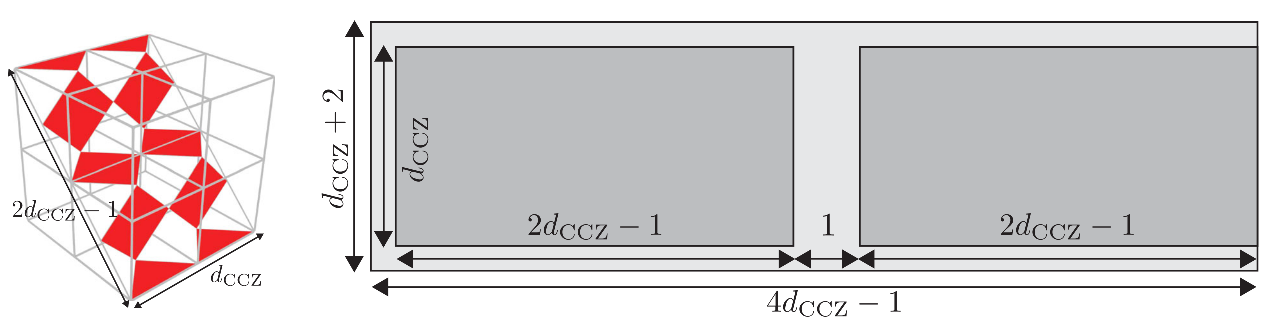
In-place gate

Besides using the linear-time CCZ gate as a way to prepare magic states without using distillation, it can also perform logical CCZ directly between any triple of surface code lattice patches in the computer by moving these patches into the correct positions and using code deformation techniques to transform the lattices as necessary. These transformations are shown in Figs. 10, 11, and 12. The transformation is simplest for the red code (Fig. 10) where we just need to prepare new qubits in $|+\rangle$ and measure additional Z stabilisers to transform a rotated surface code into the unrotated code required for the gate. For the blue code (Fig. 11) we show the transformation in two steps, first the rotated code is transformed into an unrotated code and then single-qubit Z measurements are used to transform from a square lattice to a kagome lattice. For the green code the process is similar, but the rotated code must first be transformed into a large unrotated code, then single-qubit X and Z measurements are used to obtain the target code.

Implementing logical CCZ using this in-place gate instead of magic state injection removes the cost associated with routing magic states from factories to the places they will be consumed, and also avoids computational bottlenecks that may occur if the rate of magic state consumption is higher than the rate of production.

We will write the distance of the codes used in the in-place gate as d_{CCZ}, and note that in general this can differ from the distance used when performing Clifford operations with the rotated codes. We can count the number of data qubit loops required to implement this gate by first noticing that a single layer with distance d_{CCZ} fits inside a $d_{\mathrm{CCZ}} \times d_{\mathrm{CCZ}} \times d_{\mathrm{CCZ}}$ cube, as shown in Fig. 13a. There are then d_{CCZ} data qubits along the horizontal edge of the layer and $2d_{\mathrm{CCZ}} - 1$ qubits on the diagonal edge (d_{CCZ} qubits on grid vertices plus $d_{\mathrm{CCZ}} - 1$ qubits on grid faces). As mentioned in Sec. II B, the target code patches of the CCZ gates start in two disjoint stacks, slide over each other during the application of the gate and end in disjoint stacks again. Hence, to perform the CCZ gate we need space to place two stacks end-to-end with a small space (at least one data qubit loop) between them, as shown in Fig. 13b. We also need to have a small amount of extra space (two loops in one direction and one in the other) around the edges of these patches for the transformation of the green code from rotated to unrotated, and for the boundary ancillas for the blue code. The final dimensions are then $(4d_{\mathrm{CCZ}} - 1) \times (d_{\mathrm{CCZ}} + 2)$ data qubit loops.

The full CCZ requires the codes on the left and right to switch places, so each code needs to travel $(2d_{\mathrm{CCZ}} - 1) + 1 = 2d_{\mathrm{CCZ}}$ data qubit loops to the left or right. Each step of the gate consists of three stages: expanding a layer to a slice, applying transversal physical CCZ and collapsing to the next layer, and the code will travel to left or right by one data qubit loop after one step. Thus, in total $2d_{\mathrm{CCZ}}$ steps are required for the full CCZ gate. The physical CCZ gate can be compiled using 6 CNOTs as outlined in[34]. Alternatively, there can be native implementations of CCZ (or Toffoli) in different platforms[35–39]. For



(a) (b)

Fig. 13 | Space required for implementation of linear-time CCZ in the looped pipeline architecture. a A distance d_{CCZ} layer of one code (here the red code) fits inside a $d_{\mathrm{CCZ}} \times d_{\mathrm{CCZ}} \times d_{\mathrm{CCZ}}$ cube using $2d_{\mathrm{CCZ}} - 1$ data qubits on the diagonal edge (here $d_{\mathrm{CCZ}} = 3$). **b** Code layout (measured in number of data qubit loops) needed for linear CCZ gates. The dark grey regions represent the space required for the three codes, while the extra space in the light grey region is used for the transformation of the green code from rotated to unrotated and for some of the boundary ancillas in the blue code. Here we assume that the green code is on the left while the red and blue codes are on the right.

simplicity, we will assume the gate error rate for CCZ is the same as the two-qubit error rate, since we are trying to see if linear CCZ can compete with magic state distillation using the most favourable assumption. As discussed previously, the expand stage requires 2 code cycles. The collapse stage is simply measuring out the relevant data qubits, combined with the gate time needed for the physical CCZ, the total time needed for both of these stages is roughly 1 code cycle. In this way, the total number of code cycles needed per step is $2 + 1 = 3$, and the total number of code cycles needed for performing linear CCZ is $T_{\text{lin}} = 3 \times 2d_{\text{CCZ}} = 6d_{\text{CCZ}}$.

Data availability

All data generated as part of this work can be found at https://github.com/tRowans/looped_pipeline_jit_data.

Code availability

All code used in this work can be found in the github repository located at[29].

Received: 18 February 2025; Accepted: 22 October 2025;
Published online: 28 November 2025

References

1. Bravyi, S. & König, R. Classification of topologically protected gates for local stabilizer codes. *Phys. Rev. Lett.* **110**, https://link.aps.org/doi/10.1103/PhysRevLett.110.170503 (2013).
2. Gottesman, D. The heisenberg representation of quantum computers. arXiv. http://arxiv.org/abs/quant-ph/9807006 (1998).
3. Webster, P. & Bartlett, S. D. Fault-tolerant quantum gates with defects in topological stabilizer codes. *Phys. Rev. A* **102**, https://doi.org/10.1103/PhysRevA.102.022403 (2020).
4. Bravyi, S. & Kitaev, A. Universal quantum computation with ideal clifford gates and noisy ancillas. *Phys. Rev. A* **71**, 22316 (2005).
5. Kubica, A., Yoshida, B. & Pastawski, F. Unfolding the color code. *New J. Phys.* **17**, 083026 (2015).
6. Beverland, M. E., Kubica, A. & Svore, K. M. Cost of universality: A comparative study of the overhead of state distillation and code switching with color codes. *PRX Quantum* **2**, 20341 (2021).
7. Brown, B. J. A fault-tolerant non-clifford gate for the surface code in two dimensions. *Sci. Adv.* **6**, eaay4929 (2020).
8. Vasmer, M. & Browne, D. E. Three-dimensional surface codes: Transversal gates and fault-tolerant architectures. *Phys. Rev. A* **100**, 12312 (2019).
9. Bombin, H. 2D quantum computation with 3D topological codes. arXiv. http://arxiv.org/abs/1810.09571 (2018).
10. Ryan-Anderson, C. et al. High-fidelity teleportation of a logical qubit using transversal gates and lattice surgery. *Science* **385**, 1327–1331 (2024).
11. Bluvstein, D. et al. Logical quantum processor based on reconfigurable atom arrays. *Nature* **626**, 58–65 (2024).
12. Cai, Z., Siegel, A. & Benjamin, S. Looped pipelines enabling effective 3D qubit lattices in a strictly 2D device. *PRX Quantum* **4**, 20345 (2023).
13. Scruby, T. R., Browne, D. E., Webster, P. & Vasmer, M. Numerical implementation of just-in-time decoding in novel lattice slices through the three-dimensional surface code. *Quantum* **6**, 721 (2022).
14. Kitaev, A. Y. U. Fault-tolerant quantum computation by anyons. *Ann. Phys.* **303**, 2–30 (2003).
15. Horsman, C., Fowler, A. G., Devitt, S. & Meter, R. V. Surface code quantum computing by lattice surgery. *N. J. Phys.* **14**, 123011 (2012).
16. Dennis, E., Kitaev, A., Landahl, A. & Preskill, J. Topological quantum memory. *J. Math. Phys.* **43**, 4452–4505 (2002).
17. Buonacorsi, B. et al. Network architecture for a topological quantum computer in silicon. *Quantum Sci. Technol.* **4**, 25003 (2019).
18. Boter, J. M. et al. Spiderweb array: A sparse spin-qubit array. *Phys. Rev. Appl.* **18**, 24053 (2022).
19. Kielpinski, D., Monroe, C. & Wineland, D. J. Architecture for a large-scale ion-trap quantum computer. *Nature* **417**, 709–711 (2002).
20. Lekitsch, B. et al. Blueprint for a microwave trapped ion quantum computer. *Sci. Adv.* **3**, e1601540 (2017).
21. Kaushal, V. et al. Shuttling-based trapped-ion quantum information processing. *AVS Quantum Sci.* **2**, 14101 (2020).
22. Ryan-Anderson, C. et al. Realization of real-time fault-tolerant quantum error correction. *Phys. Rev. X* **11**, 41058 (2021).
23. Hilder, J. et al. Fault-tolerant parity readout on a shuttling-based trapped-ion quantum computer. *Phys. Rev. X* **12**, 11032 (2022).
24. Draper, T. G., Kutin, S. A., Rains, E. M. & Svore, K. M. A logarithmic-depth quantum carry-lookahead adder. *Quantum Inf. Comput.* **6**, 351–369 (2006).
25. Gidney, C. & Ekerå, M. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* **5**, 433 (2021).
26. Fowler, A. G. & Gidney, C. Low overhead quantum computation using lattice surgery. arXiv http://arxiv.org/abs/1808.06709 (2019).
27. Gidney, C. & Fowler, A. G. Efficient magic state factories with a catalyzed $|CCZ\rangle$ to $2|T\rangle$ transformation. *Quantum* **3**, 135 (2019).
28. Fazio, N., Webster, M. & Cai, Z. Low-overhead magic state circuits with transversal CNOTs. arXiv. http://arxiv.org/abs/2501.10291 (2025).
29. Source code used in JIT decoding simulations. https://github.com/tRowans/JITdecoding-public.
30. Gidney, C., Shutty, N. & Jones, C. Magic state cultivation: Growing T states as cheap as CNOT gates. arXiv. http://arxiv.org/abs/2409.17595 (2024).
31. Davydova, M. et al. Universal fault tolerant quantum computation in 2d without getting tied in knots. https://arxiv.org/abs/2503.15751. 2503.15751 (2025).
32. Bauer, A. & de la Fuente, J. C. M. Planar fault-tolerant circuits for non-clifford gates on the 2d color code. https://arxiv.org/abs/2505.05175. 2505.05175 (2025).
33. Langrock, V. et al. Blueprint of a scalable spin qubit shuttle device for coherent mid-range qubit transfer in disordered Si/SiGe/SiO_2. *PRX Quantum* **4**, 20305 (2023).
34. Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, Cambridge, 2010). https://www.cambridge.org/core/books/quantum-computation-and-quantum-information/01E10196D0A682A6AEFFEA52D53BE9AE.
35. Gullans, M. J. & Petta, J. R. Protocol for a resonantly driven three-qubit toffoli gate with silicon spin qubits. *Phys. Rev. B* **100**, 85419 (2019).
36. Li, Y. et al. Quantum fredkin and toffoli gates on a versatile programmable silicon photonic chip. *npj Quantum Inf.* **8**, 1–7 (2022).
37. Goel, N. & Freericks, J. K. Native multiqubit toffoli gates on ion trap quantum computers. *Quantum Sci. Technol.* **6**, 44010 (2021).
38. Fedorov, A., Steffen, L., Baur, M., da Silva, M. P. & Wallraff, A. Implementation of a toffoli gate with superconducting circuits. *Nature* **481**, 170–172 (2012).
39. Kim, Y. et al. High-fidelity three-qubit iToffoli gate for fixed-frequency superconducting qubits. *Nat. Phys.* **18**, 783–788 (2022).
40. Additional data generated for Fig 12. https://github.com/tRowans/looped_pipeline_jit_data.

Acknowledgements

We would like to thank Sam Jaques, Mike Vasmer and Ben Brown for insightful discussions and suggestions. ZC acknowledges support from the EPSRC QCS Hub EP/T001062/1, EPSRC projects Robust and Reliable Quantum Computing (RoaRQ, EP/W032635/1), Software Enabling Early Quantum Advantage (SEEQA, EP/Y004655/1) and the Junior Research Fellowship from St John's College, Oxford. TRS acknowledges support from the JST Moonshot R&D Grant [grant number JPMJMS2061]. KN acknowledges support from JSPS KAKENHI [grant number 21H04880].

Numerical results presented in this work were partially obtained using the HPC resources provided by the Scientific Computing and Data Analysis section of the Research Support Division at OIST.

Author contributions

T.R.S. and Z.C. jointly developed the qubit shuttling schedule, T.R.S was responsible for numerical simulations and Z.C. carried out analysis of associated overheads. K.N. provided guidance on the direction of the project. T.R.S and Z.C. wrote the manuscript and all authors reviewed the manuscript.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41534-025-01133-7.

Correspondence and requests for materials should be addressed to Thomas R. Scruby, Kae Nemoto or Zhenyu Cai.