
Article:
Barber, B., Barnes, K.M., Bialas, T. et al. (10 more authors) (2025) A real-time, scalable, fast and resource-efficient decoder for a quantum computer. Nature Electronics, 8 (1). pp. 84-91. ISSN 2520-1131

https://doi.org/10.1038/s41928-024-01319-5

A real-time, scalable, fast and highly resource efficient decoder for a quantum computer

Ben Barber,[1] Kenton M. Barnes,[1] Tomasz Bialas,[1] Okan Buğdaycı,[1] Earl T. Campbell,[1,2] Neil I. Gillespie,[1] Kauser Johar,[1] Ram Rajan,[1] Adam W. Richardson,[1] Luka Skoric,[1] Canberk Topal,[1] Mark L. Turner,[1] and Abbas B. Ziad[1]

[1] *Riverlane, Cambridge, UK.*

[2] *Department of Physics and Astronomy, University of Sheffield, UK*

(Dated: September 2023)

To unleash the potential of quantum computers, noise effects on qubits' performance must be carefully managed. The decoders responsible for diagnosing noise-induced computational errors must use resources efficiently to enable scaling to large qubit counts and cryogenic operation. Additionally, they must operate at speed, to avoid an exponential slowdown in the logical clock rate of the quantum computer. To overcome such challenges, we introduce the Collision Clustering decoder and implement it on FPGA and ASIC hardware. We simulate logical memory experiments using the leading quantum error correction scheme, the surface code, and demonstrate MHz decoding speed – matching the requirements of fast-operating modalities such as superconducting qubits – up to an 881 and 1057 qubits surface code with the FPGA and ASIC, respectively. The ASIC design occupies 0.06mm^2 and consumes only 8mW of power. Our decoder is both highly performant and resource efficient, unlocking a viable path to practically realising fault-tolerant quantum computers.

I. INTRODUCTION

Quantum computers could potentially solve computational problems that are out of reach of classical computers. However, to realise this potential all architectures need to deal with the fragility of their quantum bits (qubits) [1–4]. Qubits are highly likely to interact with the environment, leading to errors. Fortunately, Quantum Error Correction (QEC) protocols enable fault-tolerant computation in the presence of noise. These protocols are based on adding redundancy, encoding and protecting information into logical qubits by using a larger number of physical qubits. While errors can still corrupt the information, a signal is periodically generated from the logical data which characterises them. A decoder running on classical hardware processes this so-called syndrome, generating as an output the inferred error that has occurred, informing the corrective steps taken in subsequent operations.

QEC must be performed continuously, creating a stream of syndrome data; as systems scale and logical error rates decrease, the amount of data that needs to be processed by a decoder increases significantly. Large computations will require real-time decoders that can process data at the rate it is received to avoid the creation of a backlog that grows exponentially with the depth of the computation [5, 6], ultimately slowing it to a halt. Superconducting quantum devices, for example, generate a round of syndrome data in less than 1μs (a rate of MHz), setting stringent requirements on decoder speed. Utility-scale quantum computers will require an optimised hardware decoder integrated in a tight loop at the heart of the control system.

Most experiments to date have used fast and accurate decoders implemented in software [7–9] to decode offline [10–13] rather than in real-time, the syndrome data being processed after the experiment has concluded. This type of decoding cannot support logic branching which is required to implement certain non-Clifford gates (the most essential gates to support quantum operations)[14]. Real-time decoding has been demonstrated in small scale experiments on ion-trap systems, using non-scalable lookup tables implemented in software that only require kHz speeds [15, 16]. However, in any scalable architecture, fast algorithmic decoders must be tightly integrated with the control system of the quantum computer to satisfy latency requirements. Decoders implemented on dedicated classical hardware, such as Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs), provide a viable path to such a solution.

To meet the challenge of developing real-time decoders, the community has begun to implement decoders on FPGAs [17–20], and provide models of implementations on ASICs [19, 21]. FPGAs will be sufficient for the medium term. They provide the flexibility to adapt and change implementations of decoders, helping to identify the parameters needed to optimise the system performance. Until recently, only small instances of surface code decoders have been implemented on FPGAs [17, 19, 20]. Promising results have recently appeared on larger examples, where decoding an 881 qubit surface code memory simulation was demonstrated in under 1μs [18] per round. However, only a toy noise model was used and the design required significant FPGA resources.

FPGA systems have high per-unit cost and power consumption, hence they are not long-term solutions for scaling to millions of qubits. Cost-effective scaling of useful quantum computers will be achieved with ASICs, which guarantee improved performance and reduced power consumption at the cost of longer development times. Tight integration between decoders and control systems in a cryogenic environment will require ASICs [22].

In this work, we introduce the Collision Clustering

(CC) decoder, designed to require few logical resources on an FPGA and low ASIC power and area occupation, while being performant enough to keep up with the syndrome generation time of the QPU. To demonstrate this, we implement CC on a Xilinx Ultrascale+ XCVU3P FPGA [23], and using industry leading EDA tools [24], we also design an implementation on a 12nm FinFET ASIC process node, signed off to a standard that is ready to be taped out. Assuming a realistic circuit-level noise model, on the FPGA we decode an 881-qubit surface code in 810ns using only 4.5% of the available computational elements (logic LUTs) and 10KB of memory. Moreover, we obtain a threshold of 0.78%. The ASIC decodes a 1057 qubit surface code in 240ns, using only 0.06mm^2 of area and 8mW of power.

The logical memory experiment simulated in this work preserves a state for a finite time period. To preserve a logical state indefinitely, the decoder must be able to handle data being streamed in. We save this investigation for future work.

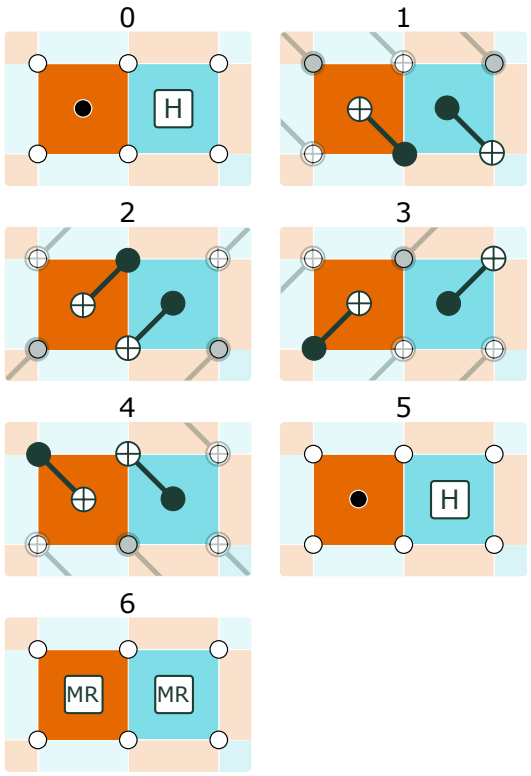


FIG. 1. Syndrome extraction circuit for a section of rotated planar code. The circuit consists of Z (orange) and X (teal) stabilizer measurements that are preformed with 4 layers of CNOT gates and two layers of single-qubit (H) gates. Finally, the ancilla qubits are measured and reset (MR) for the next round. Errors can happen at any stage of the circuit, resulting in the circuit-level noise model described in detail in Appendix C

II. THE COLLISION CLUSTERING DECODER

Collision Clustering (CC) is an implementation of Union-Find [21, 25], a decoding algorithm with "almost-linear" asymptotic scaling. The input for CC is a decoding graph (Appendix B). The vertices of this graph are all the possible defects that can occur when running the syndrome extraction circuit (Fig. 1), where a *defect* is a change in the measurement outcome of a syndrome qubit from one round of syndrome measurements to the next. The edges of the graph correspond to error mechanisms and are incident with the defects they cause. At a high level, CC decodes by first partitioning the set of defects of an input decoding graph into distinct subsets, known as clusters. Each cluster is then decoded separately using a simple procedure.

Das et al. described a detailed instruction set (micro-architecture) for implementing Union-Find on an FPGA or ASIC [21]. A key component of their micro-architecture is the spanning tree memory (STM), a data structure used in tracking the state of emerging clusters. By simulating their design we identified a bottleneck when reading from and writing into the STM, which significantly slowed down the execution of the algorithm. Therefore, we designed CC with a more memory efficient data structure to track the state of emerging clusters at the expense of asymptotic scaling. Using this architecture, on system sizes of practical interest, and larger than those modelled by Das et al. , we achieved the necessary decoding speeds.

To decode each cluster, CC uses a reference logical operator. The size of a minimal logical operator is the *distance* of a code, and is indicative of the number of errors a code can suppress. In the case of the distance d surface code, defined on a $d \times d$ square lattice of data qubits and requiring $d^2 - 1$ syndrome qubits, we use the minimal length d logical Pauli operator running along one of the boundaries of the lattice as the reference operator (Appendix A). To account for errors in the syndrome extraction circuit, certain operators are repeatedly measured giving the decoding graph a 3D structure. In this setting, the minimal logical operator is identified with a 2D boundary of the decoding graph, which we call the logical boundary. Each odd size cluster that touches this boundary flips the logical measurement, and so the correction bit returned by CC is the parity of the number of odd sized clusters that touch the logical boundary.

A. Growth and Merge of Clusters

In CC, each defect begins in its own cluster. The clusters then *grow* in the decoding graph, and, if two clusters collide, that is overlap, they *merge* to form one cluster. A cluster continues to grow so long as it has an odd number of defects within it, or until it touches one of the open boundaries of the decoding graph. Once all clusters have stopped growing, this growth-and-merge stage of the al-

gorithm terminates.

We keep track of the growing clusters in the Cluster Growth Stack (CGS) data structure (Fig. 2a). Each entry of the CGS contains the following information:

1. A *vertex_id* to represent which defect the entry is for.
2. A *growth_radius* to represent how far this defect has grown in the graph.
3. A *valid_bit*, set to 1 if this defect should be grown in the next round of growth, and set to 0 if the cluster containing it has stopped growing.

In the CGS, clusters grow by updating the growth radius of the valid entries of the stack, requiring only a single read and write operation per defect. This is significantly more memory efficient than the STM data structure proposed by Das et al. [21]. For example, for a distance $d = 15$ surface code on 449 qubits, our decoder uses 80% less memory despite being able to handle a noise model that requires more resources.

The Parent table (Fig. 2a) keeps track of which cluster a vertex belongs to. It has an address in memory for each defect, which holds the address of the corresponding parent defect. Upon initialisation, each defect is its own parent. Any entry with this property is called a *root*. When two clusters merge, we set the root of one cluster equal to the root of the other cluster, the choice of which to update being arbitrary. The Merge unit (Fig. 2a) determines whether two clusters should merge. It takes each pair of defects (one defect from each cluster), calculates the distance between them in the decoding graph, and checks if the sum of their growths is greater than this distance. If it is, the two clusters merge. For a CGS with s entries, this leads to $s^2/2$ *collision* detection comparisons.

Key to checking cluster mergers is efficient computation of the distance between two defects. We exploit the structure of the surface code to develop combinatorial functions that compute the distances without the need to traverse the graph. For a phenomenological noise model, this function consists of computing the Manhattan distance on a cubic lattice, with modifications to account for boundaries. Further modifications are required to account for extra space-time edges for the more realistic circuit-level noise model (Appendix E).

III. COLLISION CLUSTERING MICRO-ARCHITECTURE

Our micro-architecture of CC (Fig. 2a) is composed of shared memories and registers, and three processing units: Initialisation, Growth and Merge. The execution of the algorithm and the associated data structures are shown on a simple example in Fig. 2b.

A set of internal memories and registers keep the intermediary computational state. The growth of the clusters is tracked in the Cluster Growth Stack memory (Fig. 2a) as previously described. Recall also that the Parent Table (Fig. 2a) keeps track of the clusters; each defect is represented by an address in memory, and the data represents its corresponding parent defect. Three registers keep track of parameters for each growing cluster, the Boundary, Logical and Parity registers (Fig. 2a). In each, a defect is represented by an address. For the Boundary and Logical registers, the bit is set to 1 if the corresponding cluster touches any boundary, respectively the logical boundary. In the Parity register, the bit is set to 1 if there are an odd number of defects in the cluster.

Upon initialisation of the decoder, the *Init* unit (Fig. 2a) processes the decoder configuration, loads the input syndrome data and appropriate data into the storage elements.

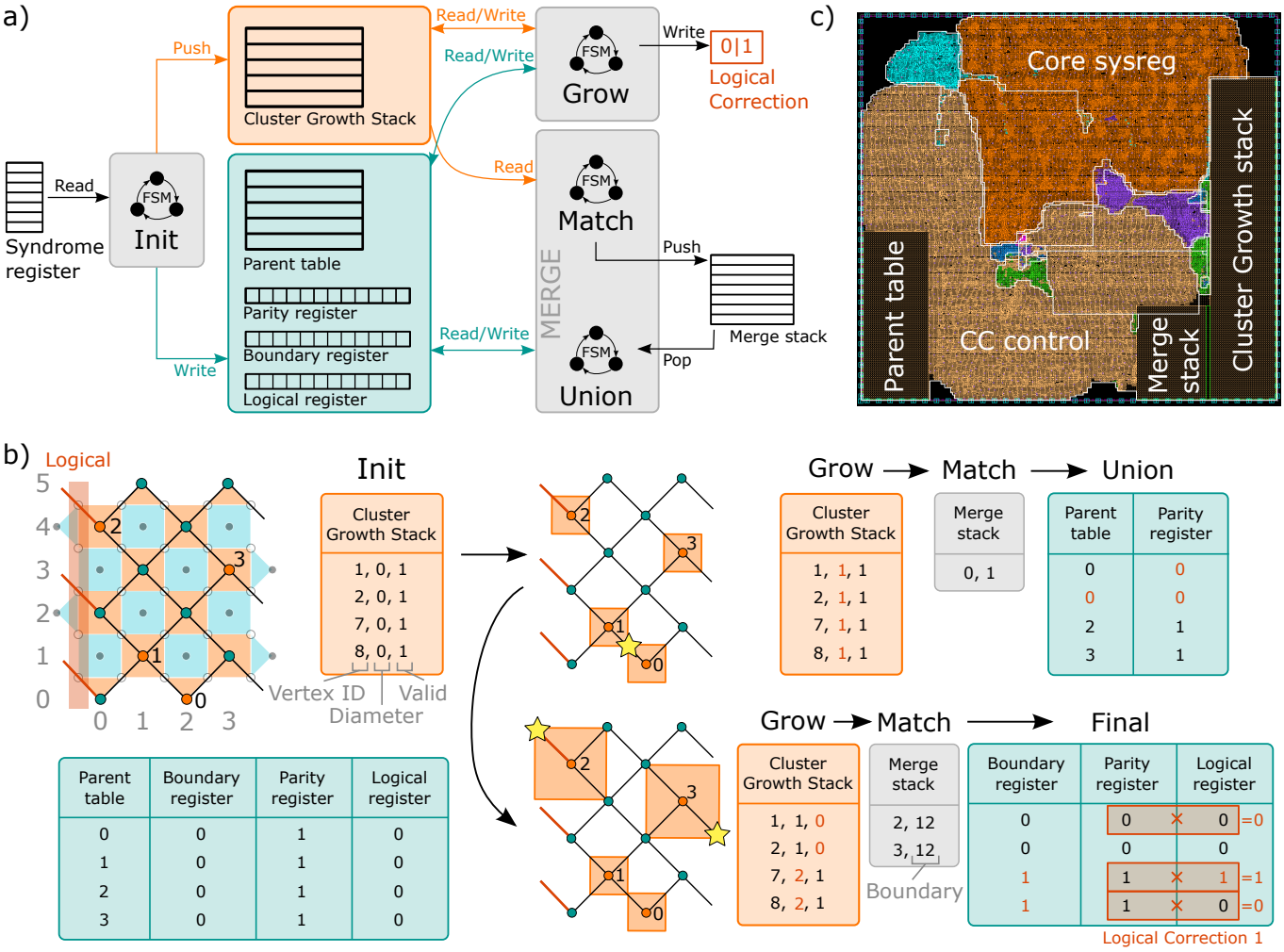
The *Grow* unit (Fig. 2a) is responsible for growing clusters. It updates the Cluster Growth Stack cluster entries at every iteration, finding the root of a cluster, then writing the radius and validity status. While processing a cluster, the Grow unit also checks for collisions with either boundary, writing the colliding vertex and the boundary or logical vertex to the Merge stack. The Grow unit also simultaneously computes the logical correction bit resulting from the growth stage. The correction is discarded and recomputed on the next growth cycle if growth is required, or is kept and used as the output correction.

The *Merge* unit consists of *Match* and *Union* sub-units (Fig. 2a), operating in parallel. The Match sub-unit performs collision detection comparisons from CGS data. It then writes colliding defect pairs onto the Merge stack. The Union sub-unit reads collided vertex entries from the Merge stack, then searches the Parent Table for the two roots. It then updates the Parent Table, Logical, Boundary and Parity registers with the results of the clusters' union.

IV. PHYSICAL IMPLEMENTATION ON FPGA

In Table I we present the performance of our FPGA implementation of CC applied to the rotated planar surface code across a range of distances from 3 (17 qubits) to 23 (1057 qubits). We use a circuit level noise model (Appendix C), specifically a depolarization channel after each 2-qubit gate with probability p, and a depolarization channel after each 1-qubit gate, measurement and reset, each with probability $p/10$. We also flip the outcome of each measurement with probability p. For the data in Table I, we set $p = 0.1\%$. We also plot in Fig. 3a the decoding time per round for varying distances and values of p. We see that the FPGA decoder can decode below the 1μs threshold up to distance 21. We targeted a maximum clock frequency (FMax) of 400 MHz, and due to the low resource utilisation, FMax is not significantly impacted by increasing the distance of the code.

The precise resource requirements on the FPGA, in-



Parent table	Boundary register	Parity register	Logical register
0	0	1	0
1	0	1	0
2	0	1	0
3	0	1	0

FIG. 2. Collision clustering decoder. (a) Micro-architecture diagram of the CC decoder computation engine, with annotated data flow. Each (sub-)unit uses a Finite State Machine (FSM) to control the computation. The input and output of the decoder is accessible through programmable registers. The logical correction bit is recalculated on the fly during Grow. The Merge processing element consists of Match and Union sub-units active in parallel. (b) Schematic of the CC decoder on an example of a single round of a rotated planar surface code. In the Init image, the decoding graph with orange-highlighted defect vertices is laid on top of the surface code with orange and teal squares, which correspond to different parity check operators. First, in the Init step, the data structures are initialised with every defect being its own parent. In the main loop, the Grow step calculates the validity of the clusters and increases the diameter of all valid clusters. The clusters are checked for collisions in the Merge step and colliding pairs of clusters (highlighted by stars in the image) are pushed to the Merge stack. The Union step pops the pairs from the Merge stack, makes one cluster root the root of the other, and updates the parity, boundary and logical registers of the root of the cluster. Boundary and logical registers are only changed when merging with the boundary – marked by auxiliary vertex 12 in the second growth step. By convention, the logical is defined to be the set of (orange) edges going to the boundary on the left side of the code and is changed to 1 when a cluster merges with the left boundary. When all clusters are invalidated, the final logical correction is calculated by summing logical registers of the roots of odd clusters. (c) The floorplan of a distance $d = 23$ (1057 qubits) ASIC implementing the CC decoder. Annotated: *Parent Table*, *Merge Stack* and *Cluster Growth Stack* SRAM cells; *CC control* logic formed of Init, Grow, and Merge processing units that are implemented as Finite State Machines (FSM). *Core sysreg* logic contains input syndrome registers, control registers and the Metric Generation Unit. Other coloured regions contain clocking, IO and other miscellaneous logic.

cluding the percentage of resource utilised, are also given in Table I. The only resources required to implement CC are trivial logic gates along with storage elements. Notably, no Digital Signal Processing (DSP) elements are needed, whose use would increase the area of the implementation.

One of the main advantages of CC is its efficient use of storage resources. Fig. 3b shows, for each code distance, the storage required for the main data structures. The micro-architecture has other storage elements e.g. Boundary, Logical and Parity registers. These are typically implemented as flip-flops and the overall results

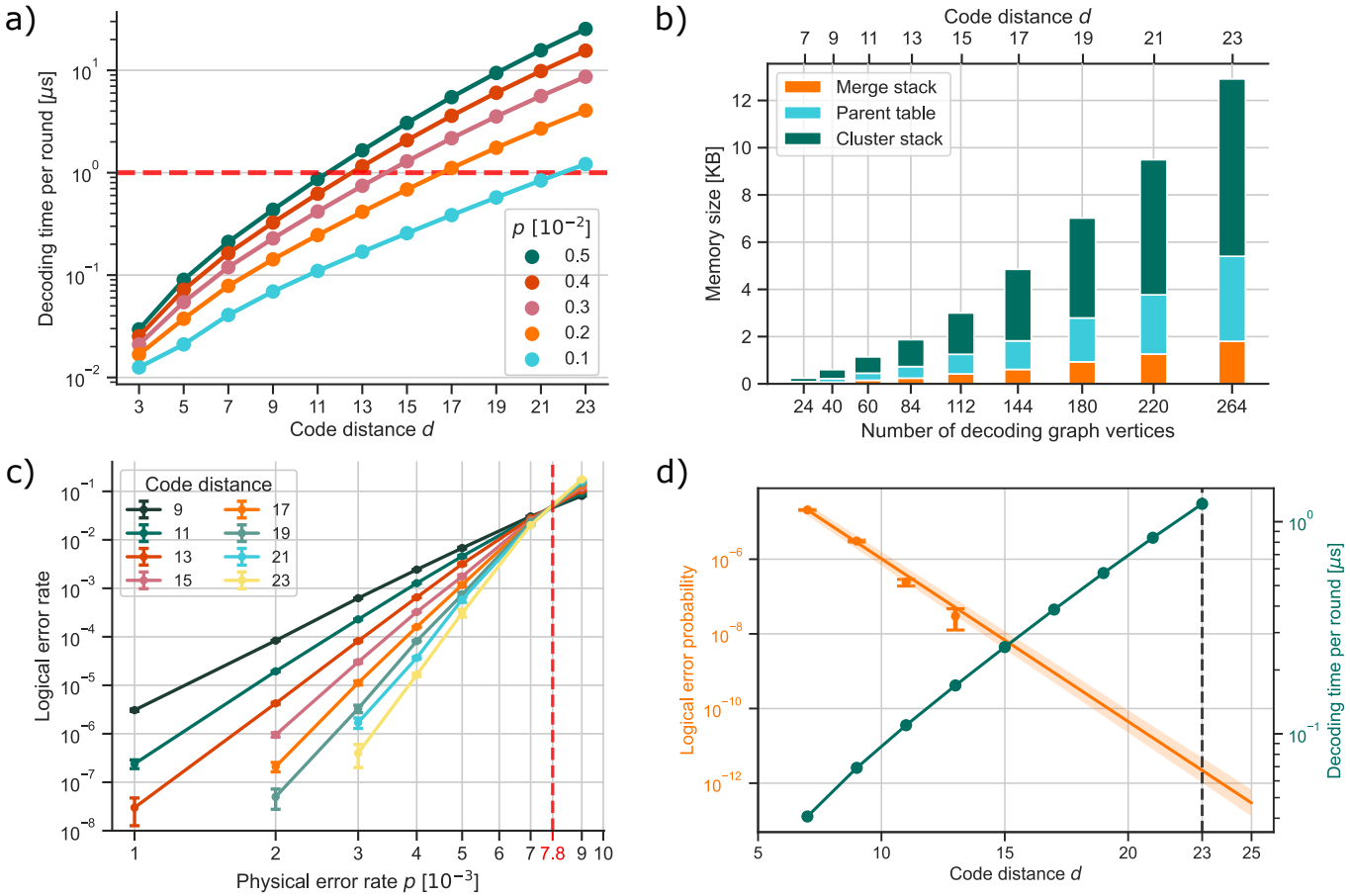


FIG. 3. Performance of our FPGA (Xilinx Ultrascale+ XCVU3P [23]) implementation of the CC decoder on the rotated planar surface code. (a) Decoding time per syndrome measurement round as a function of code distance for different noise rates. Even at a high noise just below threshold ($p = 0.5\%$) expected in near-term hardware, our decoder can decode large ($d = 11$) codes at sub-1μs per round rate. (b) FPGA memory size usage of different data structures for varying code distances using $p = 0.1\%$. Even at distance $d = 23$ the decoder uses below 13KB of memory, allowing for implementation on affordable FPGA hardware. (c) Logical error probability for varying physical error rate p and a range of code distances demonstrating threshold at around $p = 7.8 \times 10^{-3}$. Points are generated using a maximum of 10^8 samples and the points with no more than 1 error are removed. (d) Decoding time per round at $p = 0.1\%$ together with a projection of logical error probability to large distance regime. The decoder is just short of 1μs per round at distance $d = 23$ on the affordable Xilinx Ultrascale+ XCVU3P hardware while expected logical error probability is approaching 10^{-12}. Each accuracy point was obtained using 10^8 samples. Dashed vertical line is a guide to the eye. Orange line is an exponential decay fit to the data and the orange shadowed region is the standard deviation to the projection. All error bars are the standard error of the mean.

are given in Table I. For a distance 23 implementation, the main storage requirement is around 13KB which is a third of a level-1 data cache size in a typical application CPU. Similar to these caches on CPUs, the CC memories can be accessed at very high frequencies in a single clock cycle, enabling performant data processing.

Some of the other known FPGA based hardware decoders [17, 18] require substantial resources at larger code distances. Their data structures are also sized based on a phenomenological noise model at $p = 0.1\%$, so they have significantly under-counted resources compared to the requirements for a circuit-level noise model of the same magnitude, a more realistic noise model (Appendix C).

As well as being performant and resource efficient, our implementation needs to be accurate to effectively suppress physical errors. The clusters generated by CC are the same as those generated by Union-Find, resulting in the same accuracy. This intuitively holds since any cluster in Union-Find is the union of balls of different radii centered on the defects, and is confirmed empirically. We demonstrate the accuracy of CC by calculating a threshold plot [26], given in Fig. 3c. Our implementation has a threshold of 0.78%, which means that for values of p lower than this threshold errors are suppressed exponentially by increasing the code distance. To further confirm the accuracy of CC, in Fig. 3d we estimate the distances required to obtain very small logical error rates using CC. We first calculate the logical error rate for code distances up to $d = 13$ using 10^8 shots per data point. The resulting small error bars enable us to accurately project out

Code Distance	FPGA Performance		FPGA Utilisation				
	Fmax [MHz]	Exec-time [µs]	Logic LUTs	LUTRAMs	FlipFlops	RAMB36	RAMB18
3	449	0.07	2491 (0.63%)	12 (0.01%)	1572 (0.20%)	0 (0.00%)	0 (0.00%)
5	445	0.06	2709 (0.69%)	27 (0.01%)	1670 (0.21%)	0 (0.00%)	0 (0.00%)
7	406	0.06	3092 (0.78%)	49 (0.02%)	1984 (0.25%)	0 (0.00%)	0 (0.00%)
9	408	0.07	3800 (0.96%)	100 (0.05%)	2483 (0.32%)	0 (0.00%)	0 (0.00%)
11	412	0.11	4600 (1.17%)	68 (0.03%)	3591 (0.40%)	0 (0.00%)	1 (0.07%)
13	411	0.16	5914 (1.50%)	105 (0.05%)	4136 (0.52%)	0 (0.00%)	1 (0.07%)
15	403	0.25	7793 (1.96%)	60 (0.03%)	5432 (0.69%)	1 (0.14%)	1 (0.07%)
17	408	0.37	10446 (2.66%)	90 (0.05%)	7184 (0.91%)	1 (0.14%)	1 (0.07%)
19	402	0.55	13331 (3.38%)	0 (0.00%)	9277 (1.18%)	2 (0.28%)	2 (0.14%)
21	405	0.81	17237 (4.37%)	0 (0.00%)	11957 (1.52%)	2 (0.28%)	2 (0.14%)
23	401	1.18	21693 (5.50%)	0 (0.00%)	15126 (1.92%)	5 (0.69%)	1 (0.07%)

TABLE I. FPGA results for decoding the rotated planar surface code, assuming $p = 0.1\%$. Code distance: the size of the error correcting code. Fmax: the maximum achieved clock frequency on the targeted FPGA. Exec-time: the execution time averaged over 100,000 shots, normalised by dividing by the d rounds of syndrome generation. Logic LUTs: the number of FPGA lookup tables used for logic primitives. LUTRAMs, FF, RAMB36 and RAMB18: different types of storage elements. The mapping of CC data structures to a storage element depend on the size of the structure. The numbers in round bracket are the percentage of the corresponding type of resource used on the FPGA.

to the small logical error rate regime. We see that using CC, we can obtain a logical error rate approaching 10^{-12} with only a distance 23 surface code.

V. PHYSICAL IMPLEMENTATION ON ASIC

Our ASIC implementation design is ready to be taped out, having been signed-off using industry-leading Electronic Design Automation (EDA) tooling [24], top-tier foundry silicon-proven multi-Vt libraries, SRAM IP and spice models. This ensures high quality results which include all the fabrication process variations of device models and parasitic effects from the power network, clock-tree synthesis, place and route stages.

In Table II we present two physical implementations of CC on a 12nm FinFET process node: decoding a distance 7 and a distance 23 surface code. They respectively take 10ns and 240ns to decode per round of syndrome measurements, using only 2.75mW and 7.85mW of power.

The control systems used today for quantum computers cannot be scaled to control the large numbers of qubits needed to run QEC schemes that obtain low logical error rates. Cryogenic CMOS based control systems [27–29] could represent a solution, in which case only tight integration of the decoder will lead to optimal performance. Current cryogenic systems however have a strict power budget, in the order of 1W at the 4K temperature range [30]. We envisage a maximum power budget of tens of mW for a decoder, with qubit control and readout remaining the primary consumption sources. In addition to our ASIC implementations satisfying these power budget constraints, in the near term our distance 7 instance of CC will be valuable in testing error correction experiments using cryogenic CMOS based control systems.

A floorplan defines the approximate locations, sizes and shapes of various logical blocks of the design. It helps determine how signals will interact between different blocks, enabling performance, power and area optimization. The floorplan for a distance 23 implementation is shown in Fig. 2c.

VI. DISCUSSION

To the best of our knowledge, there have been four demonstrations of decoders implemented on dedicated classical hardware [17–20]. Lookup table decoders are implemented on FPGAs in [17] and [20]. In both cases, the error correction scheme demonstrated is relatively simple; the distance 3 repetition code [20], and the distance 5 surface code [17]. The exponentially scaling memory requirements make lookup table decoders impractical for surface code distances above 5. Contrary to this, we have demonstrated that the CC decoder can easily scale to handle surface code distances of practical interest.

In [19], a neural network surface code decoder is implemented on an FPGA, up to only distance 5. Measurement errors are not considered, limiting its effectiveness and understanding of how the decoder will perform with experimental qubits, counter to the more realistic noise model we use. Additionally, the corresponding estimated performance, power and area of the ASIC synthesis all degrade significantly when increasing the distance from 3 to 5, implying that the design will not effectively scale, again in contrast to the efficient scaling of CC.

The most significant prior implementation of a surface code decoder on classical hardware is found in [18]. A highly distributed implementation of Union-Find, called Helios, is implemented on an FPGA, assuming a phenomenological noise model. Each vertex of the decoding

Code Distance	ASIC Performance		ASIC Area		ASIC Power	
	Fmax [MHz]	Exec-time [µs]	Die-size [mm^2]	FlipFlops	Dynamic [mW]	Leakage [mW]
7	2000	0.01	0.009	3957	2.73	0.02
23	2000	0.24	0.064	15840	7.72	0.13

TABLE II. CC ASIC results for decoding the rotated planar surface code using a 12nm FinFET process assuming p=0.1%. We use industry-leading EDA tools to determine the frequency of the implementation. To calculate the execution time, we use this frequency along with the cycle counts of the FPGA implementations.

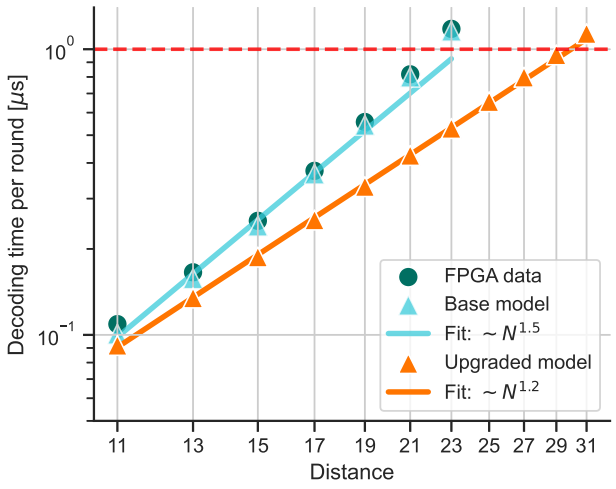


FIG. 4. Modelling improvements to the CC algorithm for the next generation FPGA decoder with $p = 0.1\%$. The results of our in-house modelling tool (base model, teal) are compared with the FPGA-acquired data (green) from Fig. 3d showing a high degree of correlation. The fit to the data is showing a $\sim N^{1.5}$ scaling as the asymptotic regime has not been reached. The modelling of the improvements to the algorithm (upgraded model, orange) demonstrates a $\sim N^{1.2}$ scaling for the whole range of distances, allowing us to stay below 1μs/round threshold up to $d = 29$. We assume 400 MHz FPGA for the modelled data.

graph is assigned a processing unit, and communication across the design is limited e.g. typically only nearest neighbour processing units can communicate with each other. In this setting, a round of syndrome measurements for a distance 21 surface code is decoded in 11.5ns on an FPGA.

The fundamental difference between Helios and CC is the amount of parallelisation that the Helios architecture can take advantage of. In Helios, all clusters are worked on simultaneously, whereas in CC, there is a limit to the amount of parallelisation that can be leveraged for performance. A consequence of this is that Helios can achieve faster decoding speeds. However, Helios' distributed network requires data transfer between neighbouring processing elements, which will always require resources. CC uses a global data structure e.g. the distance function, and optimisations of this global data structure can minimize resources. This difference is manifest when comparing the two implementations. Helios requires a large numbers of lookup tables (LUTs) and registers (900k LUTs and 240k registers). Registers are necessary in a distributed implementation so that data can be accessed at the same time in a small number of cycles. However, a register is an order of magnitude bigger in size and power consumption compared to a bit in memory. Hence, the numbers of LUTs and registers required by Helios will lead to a large area and power consumption, increasing the cost of any chip developed and preventing the integration with a cryogenic control system. Our implementation of CC is very resource efficient (Table I) while at the same time satisfies the performance requirements needed for superconducting qubits, and so is amenable to operating in cryogenic environments (Table II).

Although we have already demonstrated CC decoding at speed with low FPGA utilization for large distance codes, we are developing further improvements to enhance its performance in future generations. Currently, the Match stage performs all-to-all cluster collision checks at every growth step, a bottleneck at large distances. We can remedy this by first taking advantage of the syndrome ordering in time, reducing the number of comparisons. Secondly, as the clusters get invalidated in the later iterations of the Grow-Merge loop, most comparisons are between invalid clusters. We can avoid this by keeping track of the valid clusters and ensuring that the comparisons between invalid clusters are removed.

To quantitatively assess the impact of such improvements, we modelled the enhanced CC algorithm using a hardware-indicative Python library which predicts the number of FPGA cycles and memory footprint. This model has been successfully validated using experimental data based on the current CC algorithm implementation (Fig. 4), giving us confidence in our projections. The enhanced CC algorithm is expected to improve the scaling to $\sim N^{1.2}$; as a result, we will be able to decode a distance 29 surface code in under 1μs with only a modest sized FPGA. This is a step technological improvement with respect to [18], where the possibility to decode a distance 29 surface code in under 1μs was suggested (although not modelled), yet it would have required one of the largest commercially available FPGAs.

VII. CONCLUSION

In this work we introduced the Collision Clustering (CC) decoding algorithm and described a micro-architecture for its implementation. Fault-tolerant quantum computing requires a decoder to process error syndromes at speed in order to prevent a decoding backlog that exponentially slows down the logical clock rate. Moreover, any scalable quantum computer requires a decoder to be resource efficient, which will also enable tight integration with control systems in a cryogenic environment. To meet these requirements, we designed CC to be memory and power efficient. While CC has non-linear asymptotic scaling, this is remedied with parallelisation and pipelining for the relevant code distances, demonstrating that CC is a scalable, fast and highly resource-efficient decoder.

To verify this, we implemented CC on both an FPGA and ASIC. We decoded a logical memory experiment using large distance surface code examples in under 1μs per syndrome measurement round assuming a circuit-level noise model. On a modest sized FPGA, a distance 21 surface code took 810ns to decode per round, utilising only 4.5% of the available resources, and on a 12nm FinFET process node, a distance 23 surface code took 210ns to decode per round using only 0.06mm^2 area and 8mW power.

To preserve a logical state indefinitely, *sliding window decoding* [31] can be used. While continuous rounds of syndrome measurements are being generated, the decoder processes only a contiguous set, or *window*, of these rounds. Utilising the whole window, the decoder commits to a correction for the longer lived defects in the window, storing it in software. The window then slides up to include more recent rounds of measurements, the process repeats, and the correction is updated. Certain boundary effects make this process more complex than the one simulated in this work. Therefore, developing a fast and efficient sliding window implementation of CC will be an important next step in the advancement of decoders for fault-tolerant computation.

DATA AVAILABILITY

The stim [32] circuits used to generate the samples and raw data from all the plots in this study are available on Zenodo with the DOI identifier 10.5281/zenodo.11621877.

VIII. ACKNOWLEDGEMENTS

We thank Steve Brierley and Jake Taylor for encouraging this research and related discussions. We also thank Maria Maragkou and Luigi Martiradonna for feedback on the manuscript.

Appendix A: Surface codes

Surface codes are a family of codes that have been studied extensively since their discovery over twenty years ago [33]. They can be implemented on 2D architectures with fixed nearest neighbour interactions, a topology often used in QPUs based on superconducting qubits, for example. Moreover, extensive theoretical work has been developed to execute fault-tolerant computations based on these codes [14, 34–36]. Surface codes also achieve the highest thresholds – corresponding to the minimum qubit physical error rates required to start correcting errors effectively – among currently available error correction schemes [26], and they have been implemented in several near-term error correction experiments [10–12]. Combined, these observations make surface codes likely candidates for the error correction schemes used in the first fault-tolerant devices. In this work, we use the rotated surface code (Fig. 5) [37].

The surface code is defined on a $d \times d$ square lattice (where d is the number of data qubits in each dimension of the lattice) by operators that check the parity of sets of qubits in either the Pauli Z or X basis (Fig. 5). These operators are measured repeatedly to generate the syndrome and project the qubits into a logical computation space. We call a single round of measuring all the parity check operators a round of syndrome measurements, and the measurement data generated a round of syndrome data. For the rotated planar code, a round of syndrome measurements requires $d^2 - 1$ syndrome qubits, giving a total of $2d^2 - 1$ qubits. The logical qubit is defined by logical Pauli operators forming a path between opposite boundaries. The *distance* of the surface code is the minimum number of Pauli operators in such a logical operator, which is just the side length d of the lattice.

The quality of the QPU determines the initial error rate of the physical qubits. The distance of the surface code is a measure of the capability of this code to suppress the physical error rate down to a target logical error rate – the larger the distance, the lower the logical error rate. Quantum algorithms demonstrating industrially relevant advantage over classical computation consistently require at least 10^{12} reliable quantum operations [38–43]. Therefore, any error correction scheme needs to reduce the logical error rate to 10^{-12} or lower, which will require very large distances.

The number of physical qubits and the amount of information that needs to be processed by the decoder grow significantly with d, whereas the time available to process this information remains constant. As a result, a major challenge for the development of effective decoders is demonstrating a sufficiently fast computation even for large distances, so to avert the backlog problem.

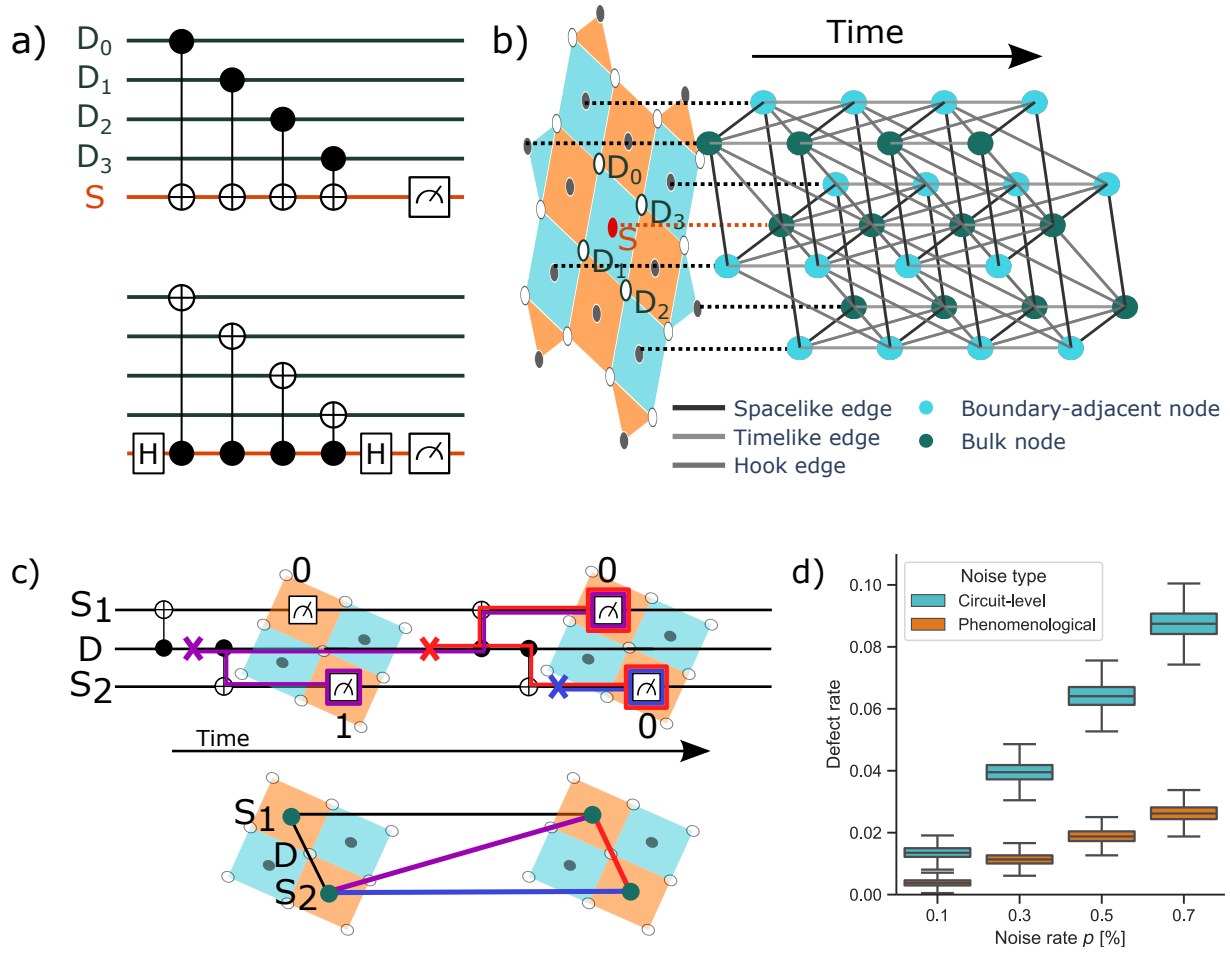


FIG. 5. Quantum Error Correction using a rotated planar surface code. (a) Quantum circuits to measure the Z (top) and X (bottom) parity checks. The circuit is continuously repeated until the computation ends and all qubits are measured out. (b) The logical information is encoded by combining multiple physical qubits laid out on a surface (orange and teal grid). The physical qubits can be divided into data qubits (D_i, empty circles) encoding the state, and syndrome qubits (S, full circles). Repeated measurements of syndrome qubits provides information about errors. Possible error mechanisms can be represented by a graph in which nodes represent differences between syndrome measurements in consecutive rounds (potential defects) and edges represent error mechanisms that create the corresponding pair of defects (Appendix B for more details). Error mechanisms that trigger only one defect (teal nodes) are in addition connected to a fictitious boundary node (not drawn). Z and X parity check patches (teal and orange squares) protect against X and Z errors respectively, and we decompose any Y error into an X error and Z error. The resulting disjoint decoding graphs are used to decode X and Z errors – only the Z check graph is shown here. (c) Two rounds of syndrome measurements displaying only a part of the circuit that involves a single data and two syndrome qubits (top). The possible error mechanisms are depicted with different colours: data (red), measurement (blue), and hook (purple) errors and result in corresponding edges in the decoding graph (bottom). (d) Comparison of defect rate for phenomenological and circuit-level noise models with the same noise rate on distance 23 rotated planar surface code. While the decoding of circuit-level noise is made more complicated by the presence of hook edges, it also results in approximately 3.5 times more defects due to more possible error locations.

Appendix B: Decoding the surface code

The core challenge in quantum error correction is to preserve a logical state, known as logical memory [10]. For the surface code, this involves initialising a logical state, performing several rounds of syndrome measurements, and finishing with a logical Pauli measurement. We are concerned with the overall effect of errors on the outcome of the logical measurement. Therefore, the decoding problem is to determine whether the logical measurement has been flipped given the observed syndrome and logical measurements.

The circuit used to measure the parity check operators (Fig. 5a) has a syndrome qubit that is reset and measured, single qubit gates, and two-qubit gates that map errors onto the syndrome qubit. In addition to the noise on the data qubits, each of these operations potentially introduce additional noise mechanisms. This is why the parity check operators are measured repeatedly, forming

the syndrome. If no errors occur, these measurements produce the same results in consecutive rounds. Therefore, an error is detected when there is a change in the outcome of a measurement from one round to the next. We call these changes in measurement outcomes *defects*.

The syndrome is best represented in a decoding graph (Fig. 5b). The vertices of the decoding graph correspond to all possible defects. If an error mechanism triggers two defects, we connect the corresponding vertices by an edge. Some error mechanisms only trigger a single defect e.g. errors on the data qubits on the boundaries of the lattice. To capture these error mechanisms, we connect the corresponding defects to virtual boundary vertices. By taking the XOR of consecutive rounds of syndrome measurements, we can identify the syndrome with the set of defects that have been triggered. The decoding problem can now be rephrased as determining the most likely logical measurement given the defects in the decoding graph generated by running the syndrome extraction circuit.

The decoding graph is actually two disjoint decoding graphs, one generated by Z checks, to correct X errors, and one generated by Z checks, to correct X errors. We decompose the Y errors included in our noise model into X and Z errors, which are handled in the appropriate decoding graph. More accurate decoding schemes exist that handle Y errors by correlating the two decoding graphs [10, 44]. We save an investigation into hardware implementations of correlated decoders for future work.

Appendix C: Noise model

Noise models vary both in the level of errors they produce as well as the types of error mechanism that can occur, which have a significant impact on the decoder performance. Throughout this work, we sample syndromes using the Clifford circuit simulator Stim [32] with several independent noise channels, parameterized by a single probability p, that give a rough approximation of noise channels characteristic of a generic superconducting device:

- Depolarisation of both qubits after each 2-qubit gate with probability p.
- Depolarisation of each idle qubit and after each single-qubit gate, including measurement and reset operations, with probability $p/10$.
- Randomly change the result of a measurement with probability p.

We use the parametrisation where depolarising a single qubit means applying a random non-identity Pauli error:

$$\mathcal{E}(\rho) = (1-p)\rho + \frac{p}{3}(X\rho X + Y\rho Y + Z\rho Z) \tag{C1}$$

Depolarising two qubits means applying one of the 15 non-identity two qubit Pauli errors uniformly at random (so that each two qubit error occurs with probability $p/15$).

Our circuit-level noise model, illustrated in Fig. 5c, generates a larger quantity and variety of defects than the phenomenological noise model [31] which abstracts away the details of generating the syndrome, and is a less realistic noise model. On a distance $d = 23$ rotated planar surface code with $p = 0.1\%$, our noise model produces defects at a rate of 1.35% (or about 3.6 defects per round), while the phenomenological noise model [18] with the same probability p produces defects at a rate of 0.38% (or about 1 defect per round, Fig. 5d). The details of the circuit that is used in our experiment are in Fig. 6 and the stim [32] circuits used to simulate the noise are available on Zenodo with the DOI identifier 10.5281/zenodo.11621877.

Appendix D: Application to other codes

In the future, different target applications will require different code distances based on the length of the computation to be performed. For this reason, decoding hardware that can only deal with a very specific code and code distance has limited value in the real world.

The decoding graph in the CC decoder is encoded by the distance calculation function, and this is the only part of the design that has to change based on the code. This makes CC and its implementations applicable to a large family of codes. The current implementation is optimised for the surface code where efficient closed form distance functions exist. For more general decoding graphs, closed-form distance functions could be developed by using graph embedding techniques, or the design could be adapted to utilise efficient lookup tables of distances.

Appendix E: Coordinate embedding and distance function

The collision clustering algorithm relies on fast and efficient calculation of the shortest path between any two nodes of the decoding graph. This can be done for a large number of cases by isometrically embedding the graph in a normed coordinate space and having a distance function compute the norm between the node coordinates. Here we outline an approach to embedding the rotated planar code graph for both the phenomenological and circuit-level noise model.

Each node can be trivially (not necessarily isometrically) embedded in a 3D space by labelling it with its coordinates:

$$X = (x_1, x_2, t) \tag{E1}$$

First, consider the phenomenological noise model and the unweighted decoding graph. In this case, there are no hook edges and all graph connections are along the principal axes of the coordinate system (see Fig. 7). The

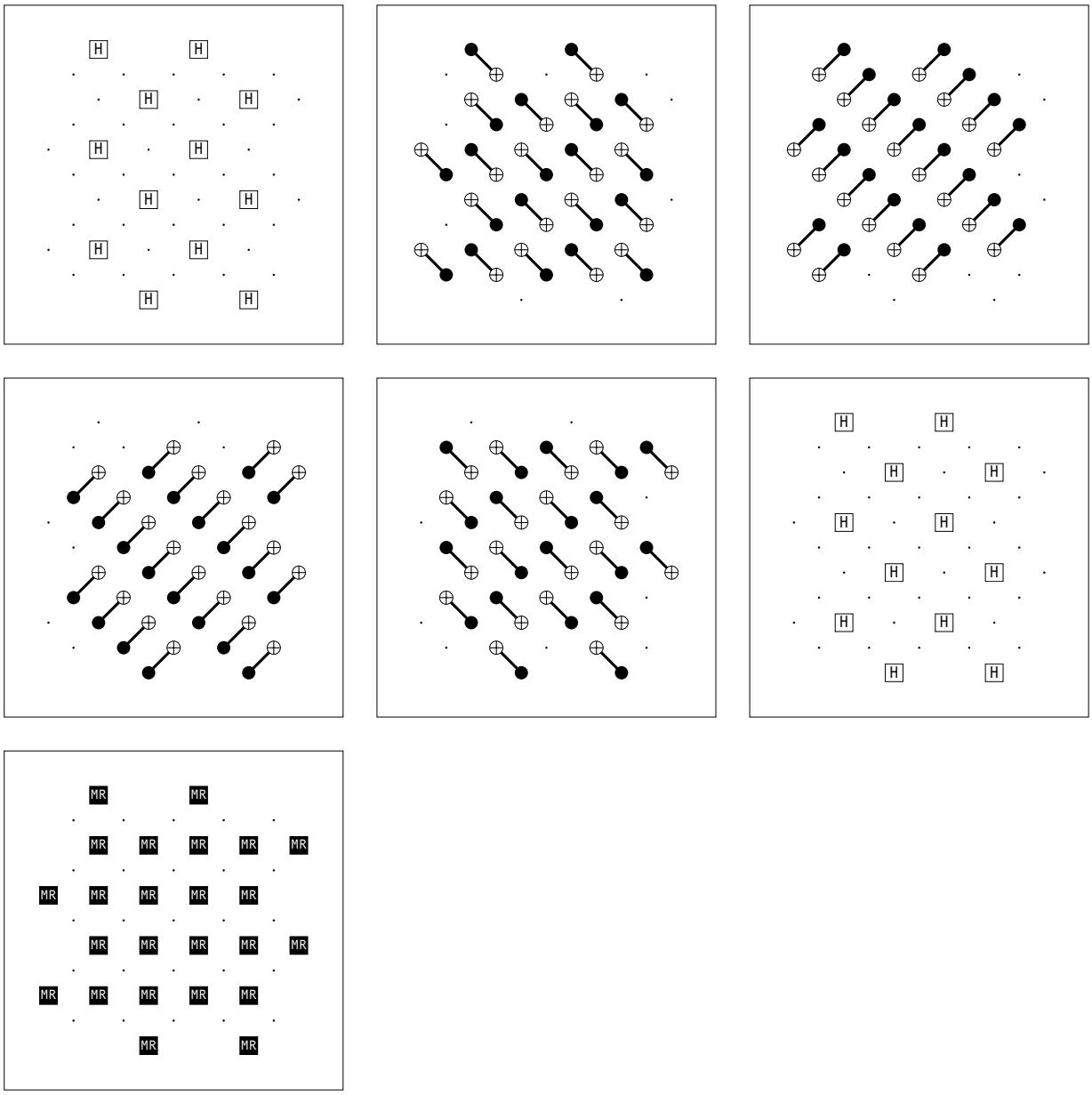


FIG. 6. Diagram of the circuit for a round of check measurements for distance 5 rotated planar code. Full circuits together with the noise model are available as stim [32] files on Zenodo with the DOI identifier 10.5281/zenodo.11621877.

distances between nodes of the graph are then computed by simply taking a Manhattan distance:

$$D(X_1, X_2) = ||X_1 - X_2||_1 = |\Delta x_1| + |\Delta x_2| + |\Delta t| \quad \text{(E2)}$$

We can also easily add weights to such a graph if we assume that the weights obey the translational symmetry (i.e., all edges along a particular direction have the same weight). In this case, we have 3 weights w_1, w_2, w_3 along the x_1, x_2, t directions respectively. The correct distance can again be calculated with the Manhattan norm, but now the embedded coordinates need to be scaled:

$$\begin{aligned} X &= (w_1 x_1, w_2 x_2, w_3 t) \quad \text{(E3)} \\ D(X_1,X_2) &= ||X_1 - X_2||_1 \\ &= w_1|\Delta x_1| + w_2|\Delta x_2| + w_3|\Delta t| \end{aligned}$$

In the circuit level noise model, hook edges are added which increases the complexity (see Fig. 7). The diagonal hook edges mean we can not isometrically embed the graph in 3D space. However, the unweighted circuit-level graph can be isometrically embedded into 4D space using

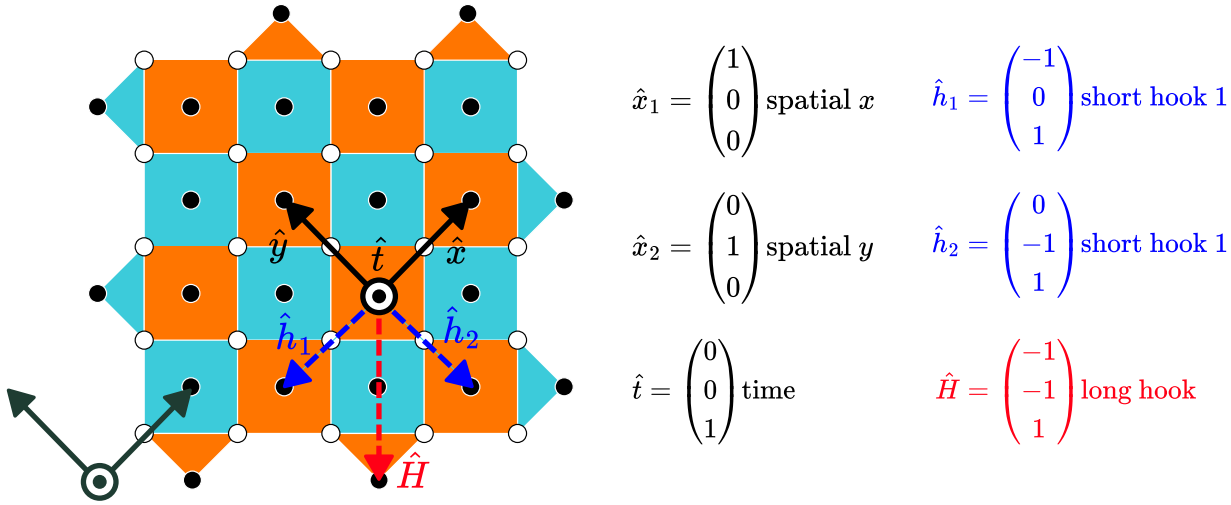


FIG. 7. Rotated planar code graph coordinate system embedding. (a) Sketch of the 5x5 rotated planar code and the principal axes. The coordinate system is aligned at 45° to the code. With the phenomenological noise model, graph is described by the nearest-neighbour connections along the $\hat{x}$, $\hat{y}$ and $\hat{t}$ directions. With the circuit-level noise model we get additional diagonal connections that are dependent on the schedule of entangling operations in the circuit and that connect nodes that are both spatially and time-like separated. In the literature, these are often referred to as hooks. Here, they are represented by $\hat{h}_1$, $\hat{h}_2$ and $\hat{H}$. A choice of the origin for the coordinate system as used in CC is shown in the bottom left corner of the figure.

the L1 norm:

$$\begin{aligned} X &= \left(\frac{x_1}{2}, \frac{x_2}{2}, \frac{x_1+t}{2}, \frac{x_2+t}{2}\right) \\ D(X_1,X_2) &= ||X_1 - X_2||_1 \\ &= \frac{1}{2}\left(|\Delta x_1| + |\Delta x_2| + \right. \\ &\quad \left. + |\Delta x_1 + \Delta t| + |\Delta x_2 + \Delta t|\right) \end{aligned} \tag{E4}$$

In the CC decoder presented in the main paper, the clusters are labelled by their coordinates (Eq. (E1)) and the distance calculated when needed according to Eq. (E4). The boundary is treated as a special node and the distance to the boundary calculated as $\min(x_1, d-x_1)$ where d is code distance and the origin of the coordinate system is as defined in Fig. 7. The Eq. (E4) can be extended to the weighted graph assuming translational symmetry, but requires an embedding in 7D space and is beyond the scope of this paper.

[1] P. Shor, in *Proceedings of 37th Conference on Foundations of Computer Science* (1996) pp. 56–65.
[2] D. Aharonov and M. Ben-Or, in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97 (Association for Computing Machinery, New York, NY, USA, 1997) p. 176–188.
[3] A. Y. Kitaev, Quantum error correction with imperfect gates, in *Quantum Communication, Computing, and Measurement*, edited by O. Hirota, A. S. Holevo, and C. M. Caves (Springer US, Boston, MA, 1997) pp. 181–188.
[4] E. Knill, Nature **434**, 39 (2005).
[5] B. M. Terhal, Rev. Mod. Phys. **87**, 307 (2015).
[6] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell, Nature Communications **14**, 7040 (2023).
[7] O. Higgott and C. Gidney, Sparse blossom: correcting a million errors per core second with minimum-weight matching (2023), arXiv:2303.15933 [quant-ph].
[8] O. Higgott, T. C. Bohdanowicz, A. Kubica, S. T. Flammia, and E. T. Campbell, Phys. Rev. X **13**, 031007 (2023).
[9] Y. Wu and L. Zhong, Fusion blossom: Fast MWPM decoders for QEC (2023), arXiv:2305.08307 [quant-ph].
[10] Google Quantum AI, Nature **614**, 676 (2023).
[11] J. Ferreira Marques, B. Varbanov, M. Moreira, H. Ali, N. Muthusubramanian, C. Zachariadis, F. Battistel, M. Beekman, N. Haider, W. Vlothuizen, A. Bruno, B. Terhal, and L. DiCarlo, Nature Physics **18**, 80 (2021).
[12] S. Krinner, N. Lacroix, A. Remm, A. Di Paolo, E. Genois, C. Leroux, C. Hellings, S. Lazar, F. Swiadek, J. Herrmann, G. J. Norris, C. K. Andersen, M. Müller, A. Blais, C. Eichler, and A. Wallraff, Nature **605**, 669 (2022).
[13] L. Postler, S. Heuβen, I. Pogorelov, M. Rispler, T. Feldker, M. Meth, C. D. Marciniak, R. Stricker, M. Ringbauer, R. Blatt, P. Schindler, M. Müller, and T. Monz, Nature **605**, 675 (2022).
[14] D. Litinski, Quantum **3**, 128 (2019).
[15] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes, and

R. P. Stutz, Phys. Rev. X **11**, 041058 (2021).
[16] M. P. da Silva, C. Ryan-Anderson, J. M. Bello-Rivas, A. Chernoguzov, J. M. Dreiling, C. Foltz, F. Frachon, J. P. Gaebler, T. M. Gatterman, L. Grans-Samuelsson, D. Hayes, N. Hewitt, J. Johansen, D. Lucchetti, M. Mills, S. A. Moses, B. Neyenhuis, A. Paz, J. Pino, P. Siegfried, J. Strabley, A. Sundaram, D. Tom, S. J. Wernli, M. Zanner, R. P. Stutz, and K. M. Svore, Demonstration of logical qubits and repeated error correction with better-than-physical error rates (2024), arXiv:2404.02280.
[17] P. Das, A. Locharla, and C. Jones, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22 (Association for Computing Machinery, 2022) p. 541–553.
[18] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, Scalable quantum error correction for surface codes using FPGA (2023), arXiv:2301.08419 [quant-ph].
[19] R. J. Overwater, M. Babaie, and F. Sebastiano, IEEE Transactions on Quantum Engineering **3**, 1 (2022).
[20] D. Ristè, L. C. G. Govia, B. Donovan, S. D. Fallek, W. D. Kalfus, M. Brink, N. T. Bronn, and T. A. Ohki, npj Quantum Information **6**, 71 (2020).
[21] P. Das, C. A. Pattison, S. Manne, D. M. Carmean, K. M. Svore, M. Qureshi, and N. Delfosse, in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022) pp. 259–273.
[22] E. Charbon, F. Sebastiano, A. Vladimirescu, H. Homulle, S. Visser, L. Song, and R. M. Incandela, in *2016 IEEE International Electron Devices Meeting (IEDM)* (2016) pp. 13.5.1–13.5.4.
[23] Xilinx ultrascale+ datasheet (2021), `https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus`. Accessed: 06/09/2023.
[24] Synopsys fusion compiler datasheet (2018), https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler.html. Accessed: 06/09/2023.
[25] N. Delfosse and N. H. Nickerson, Quantum **5**, 595 (2021).
[26] A. G. Fowler, A. M. Stephens, and P. Groszkowski, Phys. Rev. A **80**, 052312 (2009).
[27] D. J. Frank, S. Chakraborty, K. Tien, P. Rosno, T. Fox, M. Yeck, J. A. Glick, R. Robertazzi, R. Richetta, J. F. Bulzacchelli, D. Ramirez, D. Yilma, A. Davies, R. V. Joshi, S. D. Chambers, S. Lekuch, K. Inoue, D. Underwood, D. Wisnieff, C. Baks, D. Bethune, J. Timmerwilke, B. R. Johnson, B. P. Gaucher, and D. J. Friedman, in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 65 (2022) pp. 360–362.
[28] J. C. Bardin, E. Jeffrey, E. Lucero, T. Huang, S. Das, D. T. Sank, O. Naaman, A. E. Megrant, R. Barends, T. White, M. Giustina, K. J. Satzinger, K. Arya, P. Roushan, B. Chiaro, J. Kelly, Z. Chen, B. Burkett, Y. Chen, A. Dunsworth, A. Fowler, B. Foxen, C. Gidney, R. Graff, P. Klimov, J. Mutus, M. J. McEwen, M. Neeley, C. J. Neill, C. Quintana, A. Vainsencher, H. Neven, and J. Martinis, IEEE Journal of Solid-State Circuits **54**, 3043 (2019).
[29] S. J. Pauka, K. Das, R. Kalra, A. Moini, Y. Yang, M. Trainer, A. Bousquet, C. Cantaloube, N. Dick, G. C. Gardner, M. J. Manfra, and D. J. Reilly, Nature Electronics **4**, 64 (2021).
[30] S. Krinner, S. Storz, P. Kurpiers, P. Magnard, J. Heinsoo, R. Keller, J. Lütolf, C. Eichler, and A. Wallraff, EPJ Quantum Technology **6**, 1 (2019).
[31] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, Journal of Mathematical Physics **43**, 4452 (2002).
[32] C. Gidney, Quantum **5**, 497 (2021).
[33] A. Kitaev, Annals of Physics **303**, 2 (2003).
[34] C. Chamberland and E. T. Campbell, PRX Quantum **3**, 010331 (2022).
[35] A. G. Fowler and C. Gidney, Low overhead quantum computation using lattice surgery (2018), arXiv:1808.06709 [quant-ph].
[36] D. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, New Journal of Physics **14**, 123011 (2012).
[37] Y. Tomita and K. M. Svore, Phys. Rev. A **90**, 062320 (2014).
[38] C. Gidney and M. Ekerå, Quantum **5**, 433 (2021).
[39] J. Lee, D. W. Berry, C. Gidney, W. J. Huggins, J. R. McClean, N. Wiebe, and R. Babbush, PRX Quantum **2**, 030305 (2021).
[40] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, in *Post-Quantum Cryptography*, edited by J. Ding and J.-P. Tillich (Springer International Publishing, Cham, 2020) pp. 425–444.
[41] C. Gidney, M. Newman, A. Fowler, and M. Broughton, Quantum **5**, 605 (2021).
[42] N. S. Blunt, J. Camps, O. Crawford, R. Izsák, S. Leontica, A. Mirani, A. E. Moylett, S. A. Scivier, C. Sünderhauf, P. Schopf, J. M. Taylor, and N. Holzmann, Journal of Chemical Theory and Computation **18**, 7001 (2022), pMID: 36355616.
[43] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, Assessing requirements to scale to practical quantum advantage (2022), arXiv:2211.07629 [quant-ph].
[44] A. G. Fowler, Optimal complexity correction of correlated errors in the surface code (2013), arXiv:1310.0863.