arXiv:2301.08419v2 [quant-ph] 15 May 2023

Scalable Quantum Error Correction for Surface Codes using FPGA

Namitha Liyanage, Yue Wu, Alexander Deters and Lin Zhong
Department of Computer Science, Yale University, New Haven, CT
Email : {namitha.liyanage, yue.wu, alex.deters, lin.zhong}@yale.edu

***Abstract*—A fault-tolerant quantum computer must decode and correct errors faster than they appear. The faster errors can be corrected, the more time the computer can do useful work. The Union-Find (UF) decoder is promising with an average time complexity slightly higher than $O(d^3)$. We report a distributed version of the UF decoder that exploits parallel computing resources for further speedup. Using an FPGA-based implementation, we empirically show that this distributed UF decoder has a sublinear average time complexity with regard to d, given $O(d^3)$ parallel computing resources. The decoding time per measurement round decreases as d increases, a first time for a quantum error decoder. The implementation employs a scalable architecture called Helios that organizes parallel computing resources into a hybrid tree-grid structure. We are able to implement d up to 21 with a Xilinx VCU129 FPGA, for which an average decoding time is 11.5 ns per measurement round under phenomenological noise of 0.1%, significantly faster than any existing decoder implementation. Since the decoding time per measurement round of Helios decreases with d, Helios can decode a surface code of arbitrarily large d without a growing backlog.**

I. Introduction

The high error rates of quantum devices pose a significant obstacle to the realization of a practical quantum computer. As a result, the development of effective quantum error correction (QEC) mechanisms is crucial for the successful implementation of a fault-tolerant quantum computer.

One promising approach for QEC is surface codes [1–3] in which information of a single qubit (called a logical qubit) is redundantly encoded across many physical data qubits, with a set of ancillary qubits interacting with the data qubits. By periodically measuring the ancillary qubits, one can detect and potentially correct errors in physical qubits.

Once the presence of errors has been detected through the measurement of ancillary qubits, a classical algorithm, or *decoder*, guesses the underlying error pattern and corrects it accordingly. The faster errors can be corrected, the more time a quantum computer can spend on useful work. Due to the error rate of the state-of-the-art qubits, very large surface codes ($d > 25$) are necessary to achieve fault-tolerant quantum computing [2, 4, 5]. See §II for more background.

As surveyed in §VII, previously reported decoders capable of decoding errors as fast as measured, or *backlog-free*, either exploit limited parallelism [6–8], or sacrifice accuracy [9, 10]. Sparse Blossom [8] and Fusion Blossom [11] feature an important algorithmic breakthrough in realizing MWPM-based decoders. Fusion Blossom can additionally leverage measurement round-level parallelism to meet the throughput requirement of very large d. However, due to their software-based realizations, both Sparse Blossom and Fusion Blossom suffer from decoding time per round longer than that of Helios by orders of magnitude at large d and higher noise level. When used in a quantum computer, the computer would spend most of execution time waiting for error correction.

In this paper we report a *distributed Union-Find (UF) decoder* (§III) and its FPGA implementation called *Helios* (§IV). Given $O(d^3)$ parallel resources, our decoder achieves sublinear average time complexity according to empirical results for d up to 21, the first to the best of our knowledge. Notably, adding more parallel resources will not reduce the time complexity of the decoder, due to the inherent nature of error patterns. Our decoder is a distributed design of and logically equivalent to the UF decoder first proposed in [12]. We implement the distributed UF decoder with Helios, a scalable architecture for organizing the parallel computation units. Helios is the first architecture of its kind that can scale to arbitrarily large surface codes by exploiting parallelism at the vertex level of the model graph. In §VI, we present experimental validations of the distributed UF decoder and Helios using a VCU129 FPGA board [13] for up to $d = 21$. The decoder's average decoding time per measurement round under a phenomenological noise of 0.1% is 11.5 ns for $d = 21$, which is significantly faster than any existing decoder implementation. Our results successfully demonstrate, for the first time, a decoder design with decreasing average time per measurement round when d increases. This shows evidence that the decoder can scale to arbitrarily large surface codes without a growing backlog.

In summary, we report the following contributions in this work.

- A distributed algorithm that implements the Union-Find decoder that can exploit parallel computing units to stop decoding time per measurement round from growing with the code distance d.
- The Helios architecture and its FPGA-based implementation that realize the distributed Union-Find decoder.
- A set of empirical data based on the FPGA implementation that demonstrate decreasing decoding time per round as d grows and 11.5 ns decoding time per measurement round for $d = 21$ under a phenomenological noise of 0.1%.

Helios is open-source and available from [14].

II. Background

A. Error Correction and Surface Code

Quantum Error Correction (QEC) is more challenging than classical error correction due to the nature of Quantum bits. First, qubits cannot be copied to achieve redundancy due to the no-cloning theorem. Second, the value of the qubits cannot be directly measured as measurements perturb the state of qubits. Therefore QEC is achieved by encoding the *logical state* of a qubit, as a highly entangled state of many physical qubits. Such an encoded qubit is called a *logical* qubit.

The surface code is the widely used error correction code for quantum computing due to its high error correction capability and ease of implementation due to only requiring connectivity between adjacent qubits. A distance d rotated surface code is a topological code made out of $2d^2-1$ physical qubits arranged as shown in Figure 1. A key feature of surface codes is that a larger d can exponentially reduce the rate of logical errors making them advantageous. For example, even if the physical error rate is 10 times below the threshold, d should be greater than 17 to achieve a logical error rate below 10^{-10} [2].

A surface code contains two types of qubits, namely data qubits and ancilla qubits. The data qubits collectively encode the *logical state* of the qubit. The ancilla qubits (called X-type and Z-type) entangle with the data qubits and by periodically measuring the ancilla qubits, physical errors in all qubits can be discovered and corrected. An X error occurring in a data qubit will flip the measurement outcome of Z ancilla qubits connected with the data qubit and a Z error will flip the X ancilla qubits likewise. Such a measurement outcome is called *defect measurement*. Because ancilla qubits themselves could also suffer from physical qubit errors, multiple rounds of measurements are necessary. The outcomes from these multiple rounds of measurements of ancilla qubits constitute a *syndrome*. Figure 2a shows a syndrome with sample physical qubit errors and shows how they are detected by ancilla qubits. We only show X errors and measurement errors on Z-type ancillas because Z errors and measurement errors on X-type ancillas can be independently dealt with in the same way.

A syndrome can be conveniently represented by a graph called *decoding graph* in which a vertex represents a measurement outcome of an ancilla and an edge a data qubit. Vertices corresponding to defect measurements are specially marked. The weight of an edge is determined by the probability of error in the corresponding data qubit or measurement. For distance d surface code, there are $(d+1)\times(d-1)/2$ vertices. This decoding graph can be extended to three-dimensional in which multiple identical planar layers are stacked on each other. Each layer represents a round of measurement. The minimum number of measurement rounds required to complete a fault-tolerant logical operation is d, which is also the number of rounds we consider in this paper. Corresponding vertices in adjacent layers are connected by edges representing the corresponding ancilla's measurement error probability. That is, there are $(d+1)\times((d-1)/2)\times d$ vertices in this three-dimensional graph. Figure 2b shows the decoding graph for a syndrome from $d=5$ surface code.

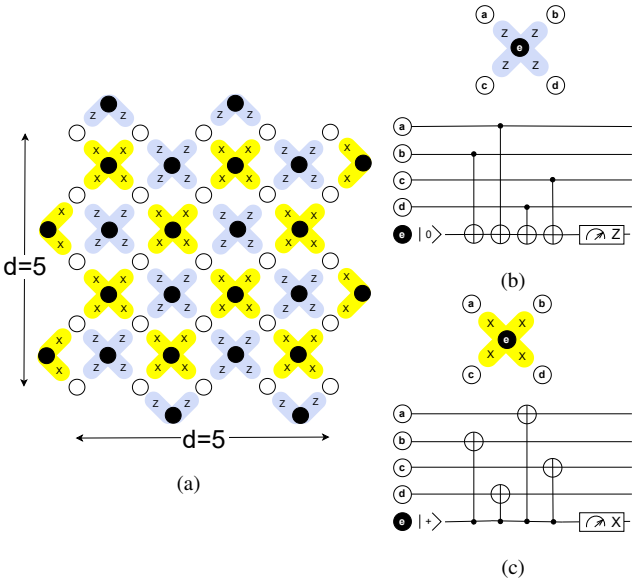


Fig. 1: (a) : Rotated CSS surface code ($d=5$), a commonly used type of surface code. The white circles are data qubits and the black are the Z-type and X-type ancillas. (b) and (c): Measurement circuit of Z-type and X-type ancillas. Excluding the ancillas in the border, each Z-type and X-type ancilla interacts with 4 adjacent data qubits.

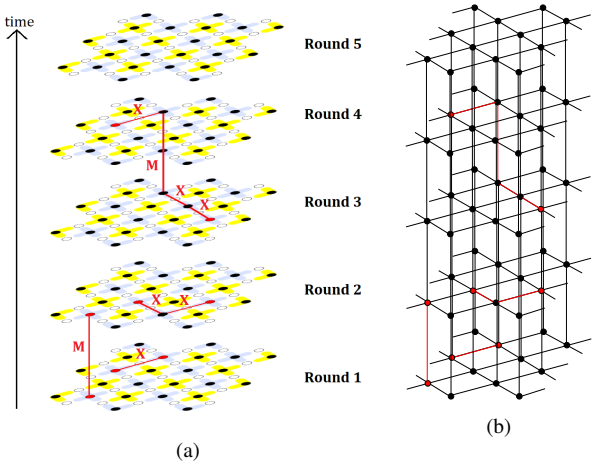


Fig. 2: (a) : An example syndrome of Z stabilizers for $d=5$ surface code with 5 rounds of measurements. The syndrome contains an isolated X-error (round 1), an isolated measurement error (rounds 1 and 2), a chain of two X errors (round 3), and a chain containing X errors and measurement errors spanning multiple measurement rounds (rounds 3 and 4). (b) Decoding graph with defect vertices marked red for the syndrome in (a).

B. Error Decoders

Given a syndrome, an error decoder identifies the underlying error pattern, which will be used to generate a correction pattern. As multiple error patterns can generate the same syndrome, the decoder has to make a probabilistic guess of the underlying physical error. The objective is that when the correction pattern is applied, the chance of the surface code

entering a different logical state (i.e a logical error) will be minimized.

a) Metrics: The two important aspects of decoders are accuracy and speed. A decoder must correct errors faster than syndromes are produced to avoid a backlog. A faster decoder also allows more time for the quantum hardware to do actual useful work. The average decoding time per measurement round is a widely used criterion for speed.

A decoder must make a careful tradeoff between speed and accuracy. A faster decoder with lower accuracy requires a larger d to achieve any given logical error rate, which may require more computation overall.

b) Union-Find (UF) Decoder: The UF decoder is a fast surface code decoder design first described by Delfosse and Nickerson [12]. According to [15], it can be viewed as an approximation to the blossom algorithm that solves minimum-weight perfect matching (MWPM) problems. It has a worst-case time complexity of $O(d^3\alpha(d))$, where α is the inverse of Ackermann's function, a slow-growing function that is less than three for any practical code distances. Based on our analysis, it has an average case time complexity slightly higher than $O(d^3)$.

Algorithm 1 describes the UF decoder. It takes a decoding graph $\mathcal{G}(\mathbf{V}, \mathbf{E})$ as input. Each edge $e \in \mathbf{E}$ has a weight and a growth, denoted by $e.w$ and $e.g$, respectively. $e.g$ is initialized with 0 and the decoder may grow $e.g$ until it reaches $e.w$. When that happens, we say the edge is *fully grown*.

The decoder maintains a set of odd clusters, denoted by $\mathcal{L}$. $\mathcal{L}$ is initialized to include all $\{v\}$ that $v \in \mathbf{V}$ are defect measurements (L5). Each cluster C keeps track of whether its cardinality is odd or even as well as its root element.

The UF decoder iterates over growing and merging the odd cluster list until there are no more odd clusters (inside the **while** loop of Algorithm 1). Each iteration has two stages: Growing and Merging. In the *Growing* stage, each odd cluster "grows" by increasing the *growth* of the edges incidental to its boundary. This process creates a set of *fully grown* edges $\mathcal{F}$ (L10 to L19). The Growing stage is the more time-consuming step as it requires traversing all the edges in the boundary of all the odd clusters and updating the global edge table. Since the number of edges is $O(d^3)$, the UF decoder is not scalable for surface codes with large d.

In the *Merging* stage, the decoder goes through each fully-grown edge to merge the two clusters connected by the edge using UNION(u, v) operation. The UNION(u, v) merges the two clusters containing u and v by assigning a common root element to the two clusters. When two clusters merge, the new cluster may become even.

When there is no more odd cluster, the decoder finds a correction within each cluster and combines them to produce the correction pattern (L25).

Algorithm 1: Union Find Decoder

```
   input : A decoding graph G(V, E) with X (or Z) syndrome
   output: A correction pattern
1  % Initialization
2  for each v ∈ V do
3      if v is defect measurement then
4          Create a cluster {v}
5      end
6  end
7  while there is an odd cluster do
8      % Growing
9      F ← ∅
10     for each odd cluster C do
11         for each e =< u, v >, u ∈ C, v ∉ C do
12             if e.growth < e.w then
13                 e.growth ← e.growth + 1
14                 if e.growth = e.w then
15                     F ← F ∪ {e}
16                 end
17             end
18         end
19     end
20     % Merging
21     for each e =< u, v >∈ F do
22         UNION(u, v)
23     end
24 end
25 Build correction within each cluster by constructing a spanning tree
```

III. Distributed UF Decoder Design

Our goal to build a QEC decoder is scalability to the number of qubits. As surface codes can exponentially reduce logical error rate with respect to d, larger surface codes with hundreds or even thousands of qubits are necessary for fault-tolerant quantum computing. Therefore, the average decoding time per measurement round should not grow with d, to avoid exponential backlog for any larger d.

We choose the UF decoder for two reasons. First, it has a much lower time complexity than the MWPM algorithm. Although in general, the UF decoder achieves lower decoding accuracy than MWPM decoders, it is as accurate in many interesting surface codes and noise models [15, 16]. Second, the UF decoder maintains fewer intermediate states, which makes it easier to implement in a distributed manner. We observe that the Growing stage from L10 to L19 in Algorithm 1 operates on each vertex independently without dependencies from other vertices. A vertex requires only the parity of the cluster it is a part of for the growing stage. Second, during the merging stage, a vertex only needs to interact with its immediate neighbors (L22).

A. Overview

Like the original UF decoder, our distributed UF decoder is also based on the decoding graph. Logically, the distributed decoder associates a processing element (PE) with each vertex in the graph. Therefore, when describing the distributed decoder, we often use PE and vertex in an inter-exchangeable manner. All PEs run the same algorithm, specified by Algorithm 2. Like the UF decoder, a PE iterates over the *Growing* and *Merging* stages with the *Merging* split into two: *Merging* and *Checking*. Within each stage, PEs operate independently. A central controller coordinates their transition from one stage to the next as specified by Algorithm 6.

A key challenge to the PE algorithm is to (*i*) merge clusters and (*ii*) compute the cluster parity, *without* central

coordination. To achieve (*i*), each PE is assigned a unique identifier (a natural number) and maintains the identifier of the cluster it belongs to, cid. The cid is the lowest identifier of all its PEs. And the PE of the lowest identifier is called the root of the cluster. When two PEs connected by a fully grown edge have different cids, the PE with the higher cid adopts the lower value, resulting in the merging of their clusters. To achieve (*ii*), each PE maintains a parent. When a PE adopts the cid from an adjacent PE, it sets the latter as its parent. The parenthood relation between PEs creates a spanning tree for each cluster that is maintained by PEs locally and in which every PE in the cluster has a directional path to the root of the cluster. The cluster parity can be computed using a convergecast algorithm on the spanning tree. We describe the PE algorithm in detail in III-D.

To implement our distributed UF algorithm, we require several PE states, some of which are located in shared memories. We limit all communication between PEs and between PEs and the controller to coherent shared memories to ensure fast communication and prevent stalling that could result from message-based communication.

B. PE States

A PE has direct read access to its local states and some states of incident PEs. A PE can only modify its local states.

Thanks to the decoding graph, a PE has immediate access to the following objects.

- v, the vertex it is associated with.
- $v.E$, the set of edges incident to v.
- $v.U$, the set of vertices that are incident to any $e \in v.E$ other than v itself. We say these vertices are adjacent to v.

The algorithm augments the data structures of each vertex and edge of the decoding graph, according to the UF decoder design [12]. For each vertex $v \in V$, the following information is added

- id : a unique identity number which ranges from 1 to n where $n = |V|$. id is statically assigned and never changes.
- m is a binary state indicating whether the measurement outcome is a defect measurement (`true`) or not (`false`). m is initialized according to the syndrome.
- cid: a unique integer identifier for the cluster to which v belongs, and is equal to the lowest id of all the vertices inside the cluster. The vertex with this lowest id is called the cluster root. cid is initialized to be id. That is, each vertex starts with its own single-vertex cluster. When $cid = id$, the vertex is a root of a cluster.
- odd is a binary state indicating whether the cluster is odd. odd is initialized to be m.
- $codd$ is a copy of odd.
- `parent` is a reference to the parent. As noted before, this parenthood relationship creates a spanning tree that connects all vertices (PEs) with directional edges.
- st_odd: a binary state representing the parity of m of v and all its descendants.
- `stage` indicates the stage the PE currently operates in

Algorithm 2: Algorithm for vertex v in the distributed UF decoder.

```
v.cid ← v.id; v.odd ← v.m; v.parent ← v.id; v.st_odd ← v.m
while true do
    if global_stage =terminate then
        return
    end
    Wait until global_stage =growing
    growing(v)
    Wait until global_stage =merging
    do
        merging(v)
        Wait until global_stage =checking
        checking(v)
        Wait until global_stage! =checking
    while global_stage =merging
end
```

Algorithm 3: Vertex growing algorithm

```
function growing(vertex v)
    v.busy ← true; v.stage ← growing
    if v.odd then
        for each e = ⟨u, v⟩ ∈ v.E atomic do
            if e.growth< e.w and u.cid ≠ v.cid then
                e.growth← e.growth+1
            end
        end
    end
    v.busy ← false;
end
```

Algorithm 4: Vertex merging algorithm

```
function merging(vertex v)
    v.busy ← true; v.stage ← merging

    for each u ∈ v.nb do
        if u.cid < v.cid then
            v.cid ← u.cid
            v.parent ← u.id
        end
    end

    v.st_odd ← XOR(u.st_odd|u ∈ v.child, m)

    if v.parent = v.id then v.odd ← v.st_odd
    else v.odd ← u.odd where v.parent = u.id

    v.busy ← false
end
```

Algorithm 5: Vertex checking algorithm

```
function checking(vertex v)
    v.busy ← true

    if ∀u ∈ v.nb, (u.cid = v.cid & v.odd = u.odd) and
      v.st_odd = XOR(w.st_odd|w ∈ v.child, m) and
      (v.parent ≠ v.id or v.odd = v.st_odd) then
        v.busy ← false
    end
    v.stage ← checking
end
```

- `busy` is a binary state indicating whether the PE has any pending operations.

Algorithm 6: The controller coordinates all PEs along stages and detects the presence of odd clusters.

```
77 while true do
78     global_stage ← growing
79     Wait until ∀v ∈ V, v.stage = growing
80     Wait until ∀v ∈ V, v.busy = false
81
82     do
83         global_stage ← merging
84         Wait until ∀v ∈ V, v.stage = merging
85         Wait until ∀v ∈ V, v.busy = false
86
87         global_stage ← checking
88         Wait until ∀v ∈ V, v.stage = checking
89     while ∃v ∈ V, v.busy = true
90
91     if ∀v ∈ V, v.codd = false then
92         global_stage ← terminate
93         return
94     end
95 end
```

For each edge $e \in E$, the decoder maintains $e.\texttt{growth}$, which indicates the growth of the edge, in addition to $e.w$, the weight. $e.\texttt{growth}$ is initialized as 0. The decoder grows $e.\texttt{growth}$ until it reaches $e.w$ and e becomes *fully grown*.

For clarity of exposition, we introduce a mathematical shorthand $v.\texttt{nb}$, the set of vertices connected with v by full-grown edges, i.e., $v.\texttt{nb}=\{u | e = \langle v, u\rangle \in v.E \ \wedge \ e.\texttt{growth}= e.w\}$. We call these vertices the *neighbors* of v. Note neighbors are always adjacent but not all adjacent vertices are neighbors. We also use $v.\texttt{child}$, to indicate all child vertices of a vertex in the tree representation, i.e., $v.\texttt{child}=\{u | u.\texttt{parent} = v.id\}$. Since trees are built within a cluster, all child vertices are neighbors but not all neighbors are child vertices.

C. Shared memory based communication

We use coherent shared memory for a shared state that has a single writer. For all shared memories, given the coherence, a read always returns the most recently written value. Like ordinary memory, we also assume both read and write are atomic. Figure 4 illustrates these memory blocks.

- memory read/write for PE (v) and read-only for adjacent PEs, i.e., $\forall u \in v.U.\ v.id$, $v.cid$, $v.odd$, $v.parent$ and $v.st\_odd$ reside in this memory (S1).
- memory read/write for PE (v) and read-only for the controller. The PE local states, $v.codd$, $v.\texttt{stage}$ and $v.\texttt{busy}$ reside in this memory (S2).
- memory for $e.\texttt{growth}$, which can be written by its two incident PEs (S3).
- memory read/write for the controller and read-only for all PEs. The controller state `global_stage` is stored in this memory (S4).

D. PE Algorithm

All PEs iterate over three stages of operation. Within each stage, they operate independently but transit from one stage to the next when the controller updates `global_stage`. When a PE enters a stage, it sets $v.stage$ accordingly and keeps $v.\texttt{busy}$ as `true` until it finishes all work in the stage. The controller uses these two pieces of information from all PEs to determine if a stage has started and completed, respectively (See §III-E).

We next describe the three stages of the PE algorithm. In the **Growing** stage, vertices at the boundary of an odd cluster increase $e.\texttt{growth}$ for boundary edges (L46). As PEs perform Growing simultaneously, two adjacent PEs may compare $e.w$ and $e.growth$ and update $e.growth$ for the same e. Such compare-and-update operations must be atomic to avoid data race.

In the **Merging** stage, two clusters connected through a fully-grown edge merge by adopting the lower cluster id (cid) of theirs. To achieve this, each PE compares its cid with its neighbors (L56). If the other incident vertex of a fully grown edge has a lower cid, the PE adopts the lower cid as its own (L57). The merging process continues until every PE in the cluster has the same cid, which is the lowest vertex identifier of the cluster.

In order to compute the cluster parity, when a PE adopts the cid of the adjacent PE, it sets the latter as its `parent` (L58). This parenthood relation creates a spanning tree for each cluster that includes all PEs (vertices) with directional edges. Each PE then calculates the parity of itself and all its children as st_odd (L65). Note that odd of the root PE is the same as its st_odd (L64). All other PEs copy the odd of their respective parents (L65).

Astute readers may point out that $v.st\_odd$ should be the parity of v and all its descendants, not just children. This is achieved by two modifications, compared to the UF decoder. First, a new stage **Checking** is added after Merging to see if the PE (vertex) needs to go back to *Merging* again (L72). Second, all PEs iterates through Merging and Checking until all PEs have nothing to do for Merging. (L34-L39). These allow parity computation to propagate from leaves to the roots of the spanning trees while cid and odd to propagate from the roots to the leaves.

a) Building corrections within clusters: While the original UF decoder builds a spanning tree within each even cluster in the end to generate a correction (L25), our distributed UF decoder already has a spanning tree based on the parenthood relation and therefore is more efficient in generating corrections.

b) Alternative Message-based Design: Early on we considered the use of message-based communication to update the parity of a cluster [17]. This design requires directional links between PEs, with each PE serving as a router for forwarding messages, thus increasing the complexity of PEs. Moreover, the finite capacity of directional links could lead to congested links, causing PE stalling, which in turn slowed down the decoding process and increased tail latency.

E. Controller Algorithm

The controller moves all PEs and itself along the three stages. In the Growing and Merging stages, it checks for $v.\texttt{busy}$ signals from each PE. The controller determines the

completion of a stage when all PEs have v.busy as false. In the Checking stage controller determines the completion of the stage when all PEs have moved to the Checking stage. Upon completion, the controller updates the global_stage variable to move to the next stage and the PEs acknowledge this update by updating their own v.stage variable.

The controller also calculates the presence of odd clusters. At the end of the Merging and Checking stages, it reads the $v.odd$ value of each vertex (L91). If any vertex has $v.odd = true$, the controller updates the global stage variable to Growing to continue the algorithm. Otherwise, it updates it to Terminate to end the algorithm.

F. Time Complexity Analysis

We first show the PE coordination complexity and then calculate the overall time complexity based on that.

a) PE Coordination Complexity: The controller's time complexity is contingent upon the implementation of the shared memory for v.busy and v.codd. Since both checks involve logical OR operations on individual PE information, the most efficient implementation consists of a logical tree of OR operations, yielding a time complexity of $O(log(d))$.

b) Worst-case Time Complexity: The worst-case time complexity of our distributed UF decoder is $O(d^3 log(d))$. We explain this as follows. Each stage of our distributed-UF algorithm is $O(1)$ time. Thus the worst case depends on the total number of stages. In the merging stage, both propagating the cid and calculating the parity uses shared memory-based flooding and convergecast algorithms, each of which requires $O(D)$ merging and checking stages, where D is the cluster diameter. The maximum possible diameter, $O(d^3)$, occurs when a series of single-vertex clusters merge, creating a chain of clusters with a total diameter of $O(d^3)$.

As coordinating between stages has a complexity of $O(log(d))$, the overall time complexity is $O(d^3 log(d))$.

Nevertheless, the worst-case scenario is extremely rare since larger clusters are exponentially less likely to occur. As shown in the empirical results reported in §VI, the average time grows sublinearly with d.

IV. Helios Architecture

We next describe Helios, the architecture for the distributed UF decoder.

A. Overview

Helios organizes PEs and the controller in a custom topology that combines a 3-D grid and a tree as illustrated by Figure 3 and explained below.

- PEs are organized according to the position of vertices in the model graph they represent. We assign $v.id$ sequentially, starting with 1 from the bottom left corner and continuing in row-major order for each measurement round. Shared memory S1 ($v.cid$, $v.odd$, $v.parent$ and $v.st\_odd$) and S2 ($v.codd$, v.stage, and v.busy) are per PE.
- Shared memory S3 (e.growth) is added to the incident PE with the lower id.
- A link between every two adjacent PEs to read from each other's S1 and for the one with the higher id to read the other's S4. This results in a network of links in a 3-D grid topology. As a PE represents a vertex in the model graph, a link represents an edge. Broad pink lines in Figure 3 represent these links.
- The controller is realized as a tree of control nodes (§IV-B). The leaf nodes of the tree contain shared memory S4.
- A link between each PE and the controller for the controller to read from S2 and for the PEs to read from S4. Dashed orange lines in Figure 3 represent these links.

B. Controller

Helios implements the controller as a tree of control nodes to avoid the scalability bottleneck. The controller requires three pieces of information from each PE: $v.codd$, v.stage and v.busy. Each leaf control node of the tree is directly connected with a subset of PEs. We can consider these PEs as the children of the leaf node. Each node in the tree gathers vertex information from its children and reports it to the parent. With information from all vertices, the root control node runs Algorithm 6 and decides whether to advance the stage.

We leave height, branching factor, and the subset of PEs connected to each leaf node as implementation choices. The necessary requirement is that the controller should not slow down the overall design.

V. FPGA Implementation

We next describe an implementation of Helios targeting a single FPGA. We choose FPGA for two reasons. It supports massively parallel logic, which is essential as the number of PEs grows proportional to d^3 in our distributed UF design. Moreover, it allows deterministic latency for each operation, which facilitates synchronizing all the PEs. Our implementation contains approximately 3000 lines of Verilog code, which is publicly available at [14].

A. Leveraging global synchronization in FPGA

We leverage global synchronization inside the FPGA to speed up our distributed UF algorithm. Running the FPGA design in a single-clock domain allows us to have all the PEs and the control nodes tightly synchronized. Notably, we simplify our algorithm as follows. Firstly, we run the Merging (L121) and Checking stages (L137) in parallel within each PE. The tight synchronization of all PEs guarantees that false negative busy signals do not occur.

Secondly, we reduce the overhead of synchronization by having the controller only coordinate moving to the Growing stage at the beginning of each iteration (L101). As each PE can perform the Growing stage deterministically in a single cycle, PEs can move to the Merging stage without central coordination (L102).

Additionally, as the controller deterministically knows the exact stage each PE is in, stage is stored locally and not shared with the controller. Thus the information from the PEs to the controller is limited to two bits, v.busy and v.odd.

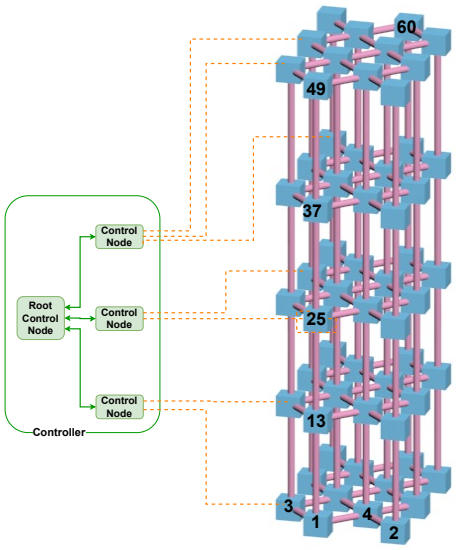


Fig. 3: Helios architecture for d=5 surface code for 5 measurement rounds. As d=5 surface code has 12 ancilla qubits of Z-type, Helios contains a 12x5 PE array. PE n indicates PE with $v.id = n$. Not all links from the controller to PEs and all $v.ids$ shown in the figure

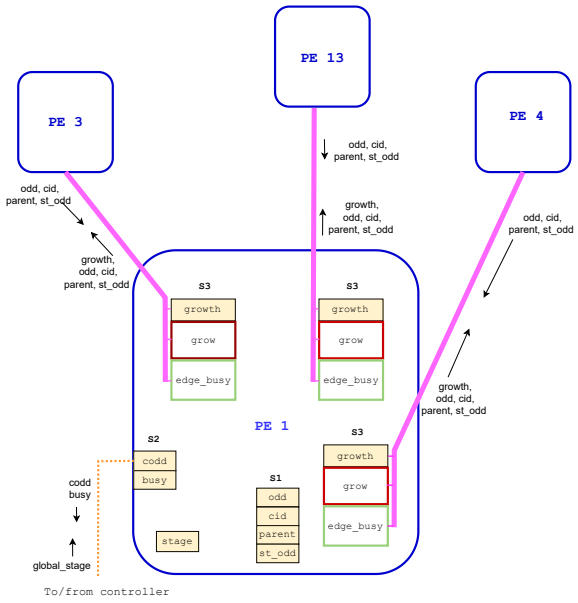


Fig. 4: The bottom left corner of the PE array shown in Figure 3. Only part of the logic and memory inside PE 1 is shown: `growth` (S3) is per edge and is stored in the PE with lower id. `grow` logic (in brown) calculates the updated growth value. `edge_busy` (in green) is per adjacent PE and is used to calculate v.`busy`.

Algorithm 7 and Algorithm 8 lists the FPGA-oriented algorithm of PE and the controller. The logic at every positive edge is executed in parallel. Figure 4 shows a minimal diagram of a PE in the FPGA implementation.

a) Time Complexity: The worst-case time complexity of the FPGA design is $O(d^3)$ in contrast to $O(d^3 log(d))$ of the generic distributed UF algorithm. The $log(d)$ factor in the latter originates from the coordination overhead associated with transitioning between Merging and Checking stages. However, in the case of FPGA implementation, these two stages—Merging and Checking—are performed concurrently, obviating stage transitions. This concurrent operation effectively removes the $log(d)$ component.

B. Implementation details

We next list the other implementation choices of our design.

Controller: Since we only use a single FPGA and evaluate with d up to 21, a single node controller suffices. The node controller reads `busy` of each PE, every clock cycle to identify the completion of a stage.

Shared memory: We implement all shared memories as FPGA registers, i.e., **`reg`** in Verilog. FPGA registers by design guarantee that a read returns the last written value. In order to ensure that the S4 memory has a single writer, we adjust the PE logic to update growth by implementing a modified compare-and-update operation (L109) as shown in Figure 5. The PE that houses the S3 memory performs this operation, increasing e.`growth` by two when both endpoints of the edge have $v.odd$ set to true.

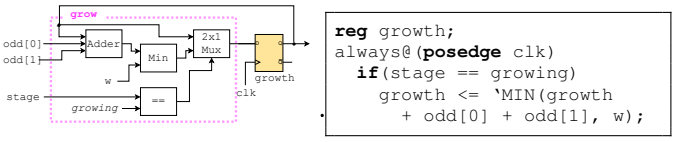


Fig. 5: Circuit diagram of `grow` sub-module and Verilog implementation. This implements the atomic compare and update operation in L45 as part of the PE module. $odd[0]$ and $odd[1]$ represents the odd state of the two incident PEs of the edge.

C. Resource Usage

On the VCU129 FPGA development board [18], we are able to support the distributed UF decoder with d up to 21, due to resource limits. Table I shows the resource usage for various d. While the numbers of vertices and edges grow by $O(d^3)$, resource usage grows faster for the following reasons. First, resource usage by a PE grows due to the increase of bit-width required for $v.id$, and $v.cid$. A PE for $d = 21$ with six adjacent PEs requires 200 LUTs and a similar PE for $d = 5$ requires only 155 LUTs. Second, PEs on the surface of the three-dimensional array as shown in Figure 3 use fewer resources than those inside because the latter have more incident edges. When d increases a higher portion of PEs are inside the array.

We find that LUTs are the most critical resource in the FPGA for our design. It may be possible to run a design with $d = 29$ on a Xilinx VU19 FPGA [19], which currently has the highest number of LUTs among commercially available FPGAs at the time of this writing. Potentially larger d values can be supported by using a network of FPGAs.

Existing commercial FPGAs like VCU129 often dedicate

Algorithm 7: FPGA-oriented algorithm for vertex v in the distributed UF decoder.

```
v.cid ← v.id; v.odd ← v.m; v.parent ← v.id;
  v.st_odd ← v.m

% Stage transition logic
At every positive clock edge do
    if global_stage =terminate then return
    else if global_stage =growing then
      v.stage ← growing
    else if v.stage =growing then v.stage ← merging
end

% Growing logic
At every positive clock edge do
    if v.stage =growing then
        for each e = ⟨u, v⟩ ∈ v.E and v.id < u.id do
            if e.growth< e.w and u.cid ≠ v.cid then
                if v.odd and u.odd then
                    e.growth← MIN(e.growth+2, w)
                end
                else if v.odd or u.odd then
                    e.growth← MIN(e.growth+1, w)
                end
            end
        end
    end
end

% Merging logic
At every positive clock edge do
    Let u be arg min_{u∈(v.nb ∪ {v})}(u.cid)
    if u.cid < v.cid then
        v.cid ← u.cid
        v.parent ← u.id
    end
end
At every positive clock edge do
    v.st_odd ← subtree_parity(v)
end
At every positive clock edge do
    if v.parent = v.id then v.odd ← v.st_odd
    else v.odd ← u.odd where u.id = v.parent
end

% Checking logic
At every positive clock edge do
    if ∃u ∈ v.nb, (u.cid ≠ v.cid || v.odd ≠ u.odd) then
        v.busy ← true
    end
    else if v.st_odd ≠ subtree_parity(v) then
        v.busy ← true
    end
    else if (v.parent = v.id & v.odd ≠ v.st_odd) then
        v.busy ← true
    end
    else
        v.busy ← false
    end
end

function subtree_parity(v)
    parity ← v.m
    for each u ∈ v.child do
        parity ← XOR(parity, u.st_odd)
    end
    return parity
end
```

Algorithm 8: FPGA-oriented controller logic

```
global_stage ← growing
At every positive clock edge do
    if global_stage = growing then
        global_stage ← merging
        %Wait until all PEs are in Merging Stage
        Wait 2 clock cycles
    end
    else if ∀v ∈ V, v.busy = false then
        if ∀v ∈ V, v.codd = false then
            global_stage ← terminate
        end
        else
            global_stage ← growing
        end
    end
end
```

TABLE I: Resource usage of Helios on VCU129 FPGA board for selected d

d	# of LUTs	# of registers
3	970	528
5	6425	2425
9	52111	13754
13	165718	47211
17	448314	122028
21	898715	238939

a lot of silicon to digital signal processing (DSP) units and block RAMs (BRAMs). However, our design does not use any DSPs because it only requires comparison operators and fixed point additions. Our design does not use any BRAMs because all communication between PEs is shared memory based, which is implemented using registers. Therefore, an ideal FPGA designed to run our distributed UF decoder would be simpler than current large FPGAs, as it would only need a large number of LUTs, no DSP units, and a limited amount of BRAM.

VI. Evaluation

The main objective of our evaluation is to assess the scalability of our distributed UF implementation. To that end, we first describe our methodology and then show that the latency of our implementation grows sub-linearly with respect to the surface code size d.

In addition, we also evaluate the impact of noise and non-identically distributed errors on latency.

A. Methodology

For speed, we measure the number of cycles required to decode a syndrome. To evaluate correctness, we compare the results of our distributed UF decoder with the results from the original UF decoder. We compare clusters because the original UF decoder and ours only differ in the clustering process. In the rest of our evaluation, we will focus only on the speed of the distributed UF decoder and not on the accuracy of its results.

a) Experimental Setup: As our evaluation setup, we use Xilinx VCU129 FPGA development board [13], which is capable of decoding surface codes with d up to 21.

We use a MicroBlaze soft processor core [20] instantiated inside the FPGA to generate the syndromes and transmit them to Helios, which runs on the same FPGA. We ran 10^6 trials for each error rate and distance.

b) Noise Model: We use the phenomenological noise model [1] that accounts for errors in both data and ancilla qubits. As decoding for X-errors and Z-errors are independent and identical, we only focus on decoding X-errors in the evaluation.

To emulate noise, we independently flip the two adjacent stabilizer measurements for each data qubit with a probability of p (the physical error rate) in each measurement round, and we also independently flip each stabilizer measurement with a probability of p except for the first and last measurement rounds. This is a widely used approach by prior QEC decoders [7, 9, 21]. We then generate the syndrome from the physical errors and provide it as input to our decoder.

For most of our experiments, we use as default $p = 0.001$, like other works [7]. This value is reasonable for surface codes, as p should be sufficiently below the threshold (at least ten times lower) to exponentially reduce errors. We note that the UF decoder has a threshold of $p = 0.024$, calculated by Delfosse and Nickerson [12].

B. Decoding Time

We experimentally show how the average time for decoding grows with the size of the surface code. Additionally, we show the effect of noise on the average time.

a) Average time: To demonstrate the scalability of our algorithm with respect to the size of the surface code, we plot the average time for decoding against the size of the surface code. In Figure 6 (left) the y-axis shows the average decoding time in nanoseconds and the x-axis shows the distance (d) of the surface code. We see that for all 3 physical error rates we tested, average decoding time grows sub-linearly with respect to the surface code size, which satisfies the scalability criteria to avoid an exponential backlog. This implies that the average time to decode a measurement round reduces with increasing d as shown in Figure 6 (right).

b) Distribution of decoding time: To understand the growth of decoding time with respect to the code distance, in Figure 7a we plot the distribution of decoding time for different code distances. The y-axis shows the decoding time and the x-axis shows the distance (d) of the surface code. The average cycle count is indicated with $\times$.

The key factor determining the decoding time is the number of iterations of growing and merging the distributed UF decoder requires. The peaks in the probability distribution for each distance in Figure 7a correspond to the number of iterations. The variation around each peak is caused by the time required to sync c_id and calculate odd. The number of iterations is related to the size of the largest cluster, which in turn correlates with the size of the longest error chain in the syndrome. As the size of the surface code increases, the probability of a longer error chain also increases, resulting in the probability distribution shifting to the right.

Furthermore, as seen in Figure 7a, the distribution for each surface code size is right-skewed. For example, for $d = 13$, 90% of trials required two iterations or fewer, which were completed within 250 ns. In the same test, 99.99% of trials were completed within 370 ns. Only a very small number of error patterns require long decoding times, corresponding to syndromes with long error chains. Since such syndromes occur rarely and have poor decoding accuracy even if the decoding time is bounded, the impact on accuracy will be minimal.

c) Effect of physical error rate: To understand the effect of the physical error rate on decoding time, in Figure 7b we plot the distribution of latency for three different noise levels for $d = 13$. The y-axis shows the latency and the x-axis shows the physical error rate.

As the noise level increases, the probability distribution of latency shifts to the right. This is caused by the increased probability of a longer error chain when the physical error rate increases, which in turn requires more iterations to decode. As a result, the average decoding time increases with the physical error rate.

C. Non-identically distributed errors

We next analyze the decoding process of a surface code with varying error probabilities for data and measurement qubits. While identically distributed errors are useful for evaluating the decoder's performance, practical implementation of surface codes may have different error probabilities for each qubit. To address this issue, each edge i in the decoding graph is assigned a weight w_i that ranges from 2 to w_{max} and is proportional to $-log(p_i)$, where p_i is the error probability corresponding to edge i. w_{max} is a user-specified parameter indicating the resolution of error probabilities.

Noise model : We assign random error probabilities from a standard normal distribution with a mean of 0.001 and a standard deviation of 0.0005.

Figure 7c shows that the average latency increases as w_{max} increases. When the errors have a higher resolution, more iterations are required for each cluster, leading to an increase in latency. For the unweighted graph with $d = 13$, the average decoding time per round of 15 ns increases to 38 ns when w_{max} increases to 16. Notably, all of these values are significantly faster than the rate of measurement. As a result, decoding non-identically distributed errors can be performed in real-time using distributed UF on Helios.

D. Comparison with related work

Our empirical results as shown in Figure 7a suggest that Helios has a lower asymptotic complexity than any existing MWPM or UF implementation for which asymptotic complexities are available, e.g., [12, 22]. Indeed, the empirical results suggest that our decoder has a sub-linear time complexity: the decoding time per round decreases with the number of measurement rounds, which has never been achieved before. This implies that Helios can support arbitrarily large d as the rate of decoding will always be faster than the rate of measurement.

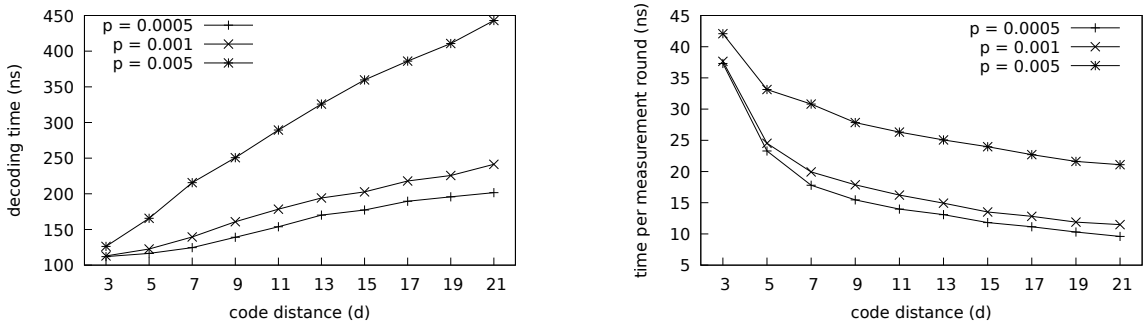
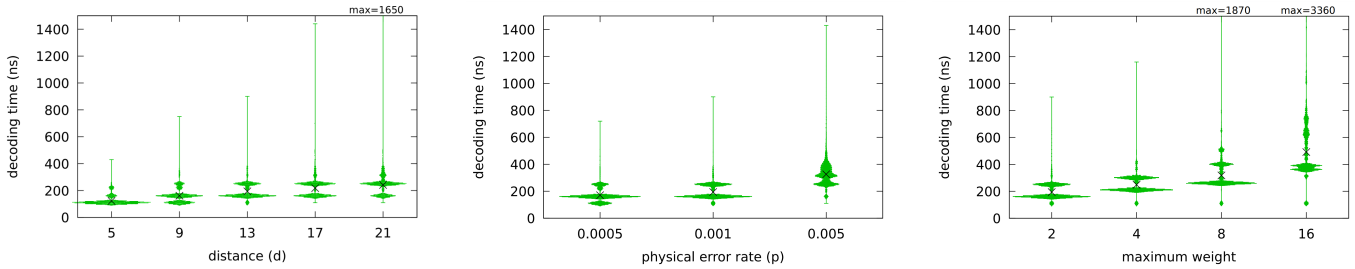


Fig. 6: Average decoding time scales sub-linearly with d. We measure the average decoding time for 3 different noise levels. (Left) The average decoding time. (Right) The average decoding time per measurement round. The average time per measurement round reducing continuously justifies that our decoder is scalable for large surface codes. We show the distributions separately in Figure 7a.



(a) T's distribution has a small mean & a long tail (b) T grows with the physical error rate. (c) T grows with the weight of the edges.

Fig. 7: Distribution of decoding time (T) with the mean marked with $\times$. Each distribution includes 10^6 data points. By default $d = 13$, $p = 0.001$ and is unweighted

Das et al [7] calculate an average latency for their AFS decoder based on memory access cycles and assuming a design running at 4 GHz. As the number of memory access cycles grows quadratically with d, the average decoding time per measurement round of AFS grows $O(d^2)$. Similarly, Ueno et al [10] estimate the decoding time of QECOOL from $d = 5$ to $d = 13$ based on SPICE-level simulations with a clock frequency of 5 GHz. For the given range of d, the decoding time per measurement round increases quadratically with d. In comparison, the decoding time of Helios decreases per measurement round.

We should like to point out that AFS and QECOOL assume very high clock frequencies, which is key to their estimated low latency. For example, for $d = 11$, AFS and QECOOL respectively report latencies of 42 ns and 8.32 ns per measurement round. Helios, in contrast, requires 16.2 ns per measurement round with a 100 MHz clock.

To the best of our knowledge, LILLIPUT [6] is the only hardware decoder in the literature that provides implementation-based results, for $d = 5$. The decoder has an average time of 21 ns per measurement round, which is slightly lower than that of Helios for $d = 5$, i.e., 24.5 ns. However, as analyzed in §VII, LILLIPUT is not scalable for $d > 5$. Our work, in contrast, has successfully demonstrated the implementation of a $d = 21$ surface code on a VCU129 FPGA with 11.5 ns per measurement round. The architecture of Helios can potentially support larger d using a larger FPGA, for example, $d = 29$ for Xilinx VU19P [19], and even larger d using a network of FPGAs.

Our decoder outperforms the two fastest software MWPM decoder, Sparse Blossom [8] and Fusion Blossom [11], by an order of magnitude. According to our evaluation, Sparse Blossom and Fusion Blossom take 160 ns and 295 ns per measurement round, respectively, for $d = 13$ and $p = 0.1\%$, using a single core of an M1 Max processor. In contrast, Helios achieves an average decoding time of 15 ns per measurement round under the same conditions, which is more than 60 times faster than the current state-of-the-art measurement rate [4].

VII. Related Work

There is a large body of literature on fast QEC decoding, e.g., [23–26]. The most related are solutions that leverage parallel compute resources.

Fowler [22] describes a method for decoding at the rate of measurement ($O(d)$). The proposed design divides the decoding graph among specialized hardware units arranged in a grid. Each unit contains a subset of vertices and can independently decode error chains contained within it. The design is based on the observation that large error patterns spanning multiple units are exponentially rare, so inter-unit communication is not frequently required. It, however, paradoxically assumes that the number of vertices per unit is "sufficiently large" and a unit can find an MWPM for its vertices within half the measurement time on average. Not surprisingly, to date, no implementation or empirical data have been reported for

this work. Our approach uses vertex-level parallelism and leverages the same observation that communication between distant vertices is infrequent.

NISQ+[9] and QECOOL[10] parallelize computation at the ancilla level, where all vertices in the decoding graph representing measurements of one ancilla are handled by a single compute unit. This results in an increase in decoding time per measurement round as d increases. In contrast, we allocate a processing element per each vertex, which results in decreasing decoding time per measurement round with d at the expense of the number of parallel units growing $O(d^3)$. Furthermore, they both implement the same greedy decoding algorithm that has much lower accuracy than the UF decoder used in this work. QECOOL has an accuracy that is approximately four orders of magnitude lower than that of a UF decoder [7] and NISQ+ ignores measurement errors further lowering its accuracy than QECOOL.

Skoric et al. [21], Tan et al. [27] and Wu [11] propose similar methods of using measurement round-level parallelism, in which a decoder waits for a large number of measurement rounds to be completed and then decodes multiple blocks of measurement rounds in parallel. By using sufficient parallel resources these methods can achieve a rate of decoding faster than the rate of measurement. However, the latency of such approaches grows with the number of measurement rounds the decoder needs to batch to achieve a throughput equal to the rate of measurement. In contrast, our approach exploits vertex-level parallelism and completes the decoding of every d round of measurements with an average latency that grows sublinearly with d.

Pipelining can be considered a special form of using compute resources in parallel, i.e., in different pipeline stages. AFS [7] is a UF decoder architected in three pipeline stages. The authors estimate the decoder will have a 42 ns latency for $d = 11$ surface code, which is 2.4 times higher than what we report based on implementation and measurement. The authors assume specialized hardware that is capable of running at 4 GHz and as a result, the decoding latency will be dominated by memory access. However, no implementation or cycle-accurate simulation is known for this decoder. Importantly, pipelining is limited in how much parallelism it can leverage: the number of pipeline stages. In contrast, the parallelism of our decoder grows along d^3, which enables us to achieve a sublinear average case latency.

LILLIPUT [6] is a three-stage look-up-table based decoder similar to AFS. Look-up-table based decoders can achieve fast decoding but are not scalable beyond $d = 5$ as the size of the look-up table grows $O(2^{d^3})$. For $d = 7$ surface code with 7 measurement rounds, it would require a memory of 2^{168} Bytes, which is infeasible in any foreseeable future.

Sparse Blossom [8], a C++ MWPM implementation, decodes faster than the rate of measurement for $d = 17$ on a single CPU core. However, its decoding time per round grows linearly with d and increases to a few micro-seconds when the noise level increases, making it impractical for real-time decoding for higher noise levels and large surface codes. Fusion Blossom [11] takes a similar approach to Sparse Blossom and additionally parallelizes the computation at the measurement round level. By allocating 100 measurement rounds to each core on a 64-core processor, Fusion Blossom can decode up to $d = 33$ faster than the measurement rate. However, both Fusion blossom and Sparse Blossom has a decoding time per round higher than that of Helios by orders of magnitude, which limits their immediate use in quantum computing.

VIII. Conclusion

We describe a distributed design for the Union Find decoder for quantum error-correcting surface codes, along with Helios, a system architecture for its realization. Our FPGA-based implementation of Helios demonstrates empirically that the average decoding time grows sub-linearly with the d. Using a VCU129 FPGA, Helios decodes distance 21 surface codes at an average speed of 11.5 ns per measurement round, the fastest to the best of our knowledge. Helios is faster and more scalable than any previously reported surface code decoder implementations. Our results suggest that by leveraging parallel hardware resources, Helios can avoid a growing backlog of measurements for arbitrarily large surface codes.

Acknowledgments

This work was supported in part by Yale University and NSF MRI Award #2216030.

REFERENCES

[1] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, 2002.

[2] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, p. 032324, 2012.

[3] J. P. Bonilla-Ataides, D. K. Tuckett, S. D. Bartlett, S. T. Flammia, and B. J. Brown, "The XZZX surface code," *arXiv e-prints*, pp. arXiv–2009, 2020.

[4] Z. Chen, K. J. Satzinger, J. Atalaya, A. N. Korotkov, A. Dunsworth, D. Sank, C. Quintana, M. McEwen, R. Barends, P. V. Klimov, S. Hong, C. Jones, A. Petukhov, D. Kafri, S. Demura, B. Burkett, C. Gidney, A. G. Fowler, H. Putterman, I. Aleiner, F. Arute, K. Arya, R. Babbush, J. C. Bardin, A. Bengtsson, A. Bourassa, M. Broughton, B. B. Buckley, D. A. Buell, N. Bushnell, B. Chiaro, R. Collins, W. Courtney, A. R. Derk, D. Eppens, C. Erickson, E. Farhi, B. Foxen, M. Giustina, J. A. Gross, M. P. Harrigan, S. D. Harrington, J. Hilton, A. Ho, T. Huang, W. J. Huggins, L. B. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, K. Kechedzhi, S. Kim, F. Kostritsa, D. Landhuis, P. Laptev, E. Lucero, O. Martin, J. R. McClean, T. McCourt, X. Mi, K. C. Miao, M. Mohseni, W. Mruczkiewicz, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Newman, M. Y. Niu, T. E. O'Brien, A. Opremcak, E. Ostby, B. Pató, N. Redd, P. Roushan, N. C. Rubin, V. Shvarts, D. Strain, M. Szalay, M. D. Trevithick, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, S. Boixo, V. Smelyanskiy, Y. Chen, A. Megrant, and J. Kelly, "Exponential suppression of bit or phase errors with cyclic error correction," *Nature*, vol. 595, no. 7867, pp. 383–387, Jul 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-03588-y

[5] C. Gidney and M. Ekerå, "How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, p. 433, Apr 2021. [Online]. Available: http://dx.doi.org/10.22331/q-2021-04-15-433

[6] P. Das, A. Locharla, and C. Jones, "LILLIPUT: A lightweight low-latency lookup-table based decoder for near-term quantum error correction," 2021. [Online]. Available: https://arxiv.org/abs/2108.06569

[7] P. Das, C. A. Pattison, S. Manne, D. M. Carmean, K. M. Svore, M. Qureshi, and N. Delfosse, "Afs: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 259–273.

[8] O. Higgott and C. Gidney, "Sparse blossom: correcting a million errors per core second with minimum-weight matching," 2023.

[9] A. Holmes, M. R. Jokar, G. Pasandi, Y. Ding, M. Pedram, and F. T. Chong, "NISQ+: Boosting quantum computing power by approximating quantum error correction," 2020. [Online]. Available: https://arxiv.org/abs/2004.04794

[10] Y. Ueno, M. Kondo, M. Tanaka, Y. Suzuki, and Y. Tabuchi, "QECOOL: On-line quantum error correction with a superconducting decoder for surface code," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2021.

[11] "Fusion Blossom," https://github.com/yale-paragon/fusion-blossom, 2023.

[12] N. Delfosse and N. H. Nickerson, "Almost-linear time decoding algorithm for topological codes," *arXiv preprint arXiv:1709.06218*, 2017.

[13] Xilinx, "Zynq UltraScale+ RFSoC ZCU106 evaluation kit," https://www.xilinx.com/products/boards-and-kits/zcu106.html

[14] "Helios scalable QEC," https://github.com/yale-paragon/Helios_scalable_QEC, 2023.

[15] Y. Wu, N. Liyanage, and L. Zhong, "An interpretation of union-find decoder on weighted graphs," 2022. [Online]. Available: https://arxiv.org/abs/2211.03288

[16] S. Huang, M. Newman, and K. R. Brown, "Fault-tolerant weighted union-find decoding on the toric code," *Physical Review A*, vol. 102, no. 1, Jul 2020. [Online]. Available: http://dx.doi.org/10.1103/PhysRevA.102.012419

[17] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, "Scalable quantum error correction for surface codes using FPGA," 2023. [Online]. Available: https://arxiv.org/abs/2301.08419

[18] Xilinx, "Virtex UltraScale+ 56G PAM4 VCU129 FPGA evaluation kit," https://www.xilinx.com/products/boards-and-kits/vcu129.html

[19] Xilinx, "Virtex UltraScale+ VU19P FPGA," https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/virtex-ultrascale-plus-vu19p-product-brief.pdf

[20] Xilinx, "MicroBlaze processor quick start guide," https://docs.xilinx.com/v/u/en-US/microblaze-quick-start-guide-with-vitis

[21] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell, "Parallel window decoding enables scalable fault tolerant quantum computation," 2022. [Online]. Available: https://arxiv.org/abs/2209.08552

[22] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time," 2014.

[23] F. Battistel, C. Chamberland, K. Johar, R. W. J. Overwater, F. Sebastiano, L. Skoric, Y. Ueno, and M. Usman, "Real-time decoding for fault-tolerant quantum computing: Progress, challenges and outlook," 2023.

[24] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, vol. 87, no. 2, pp. 307–346, apr 2015. [Online]. Available: https://doi.org/10.1103%2Frevmodphys.87.307

[25] D. Gottesman, "An introduction to quantum error correction and fault-tolerant quantum computation," 2009. [Online]. Available: https://arxiv.org/abs/0904.2557

[26] H. Bombín, *Topological codes*. Cambridge University Press, 2013, p. 455–481.

[27] X. Tan, F. Zhang, R. Chao, Y. Shi, and J. Chen, "Scalable surface code decoders with parallelization in time," 2022.