IEEE Transactions on Quantum Engineering

Received 14 February 2024; revised 16 September 2024; accepted 16 September 2024; date of publication 25 September 2024; date of current version 24 October 2024.

Digital Object Identifier 10.1109/TQE.2024.3467271

FPGA-Based Distributed Union-Find Decoder for Surface Codes

NAMITHA LIYANAGE, YUE WU, SIONA TAGARE, AND LIN ZHONG

Department of Computer Science, Yale University, New Haven, CT 06511 USA

Corresponding author: Namitha Liyanage (e-mail: namitha.liyanage@yale.edu).

A prior version of this work appeared in [DOI: 10.1109/QCE57702.2023.00106].

This work was supported in part by Yale University and in part by the National Science Foundation through the Major Research Instrumentation Program under Award 2216030.

ABSTRACT A fault-tolerant quantum computer must decode and correct errors faster than they appear to prevent exponential slowdown due to error correction. 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 a field-programmable gate array (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, the 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. Using a Xilinx VCU129 FPGA, we successfully implement d up to 21 with an average decoding time of 11.5 ns per measurement round under 0.1% phenomenological noise and 23.7 ns for $d = 17$ under equivalent circuit-level noise. This performance is significantly faster than any existing decoder implementation. Furthermore, we show that Helios can optimize for resource efficiency by decoding $d = 51$ on a Xilinx VCU129 FPGA with an average latency of 544 ns per measurement round.

INDEX TERMS Field-programmable gate array (FPGA), quantum error correction (QEC), surface codes, Union-Find (UF).

I. INTRODUCTION

The high error rates of quantum devices pose a significant obstacle to realizing a practical quantum computer. As a result, developing effective quantum error correction (QEC) mechanisms is crucial for successfully implementing a fault-tolerant quantum computer.

One promising approach for QEC is surface codes [1], [2], [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. One can detect and potentially correct errors in physical qubits by periodically measuring the ancillary qubits.

Once errors have been detected by measuring 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 state-of-the-art qubits, very large surface codes ($d \approx 27$) are necessary to achieve fault-tolerant quantum computing [2], [4], [5]. Here, d is the distance of the code and is the minimum number of bit or phase flips of physical qubits needed to change the state of the logical qubit. See Section II for more background.

As surveyed in Section VII, previously reported decoders capable of decoding errors as fast as measured, or *backlog-free*, either exploit limited parallelism [6], [7], [8], [9], [10] or sacrifice accuracy [11], [12], [13]. Sparse Blossom [8] and Fusion Blossom [14] feature an important algorithmic breakthrough in realizing minimum-weight perfect matching (MWPM)-based decoders. Fusion Blossom can additionally leverage measurement round-level parallelism to meet the throughput requirement of very large d. Due to their software-based implementations, Sparse Blossom and Fusion Blossom suffer from decoding times per round that are orders of magnitude longer than this work, especially at larger d and higher noise levels. When used in a quantum computer, the computer would spend most of the execution time waiting for error correction results.

In this article, we report a distributed Union-Find (UF) decoder (see Section III) and its field-programmable gate array (FPGA) implementation called Helios (see Section 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 decoder's time complexity 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 [15].

We implement the distributed UF decoder using Helios, a scalable architecture that efficiently organizes parallel computation units. Helios also allows for a customizable balance between latency and resource usage, adapting to specific requirements. 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 Section VI, we present experimental validations of the distributed UF decoder and Helios using a VCU129 FPGA board [16] for various values of d up to 51. When optimized for decoding time, the decoder can decode d up to 21 for phenomenological noise and d up to 17 for circuit-level noise. The average decoding time per measurement round under 0.1% noise level is 11.5 and 21.3 ns, respectively. When optimized for resource usage, the decoder can decode d up to 51, with an average decoding time of 543.9 ns per measurement round under phenomenological noise of $p = 0.001$ for $d = 51$. These results show that our decoder is significantly faster than any existing decoder implementation. Our results also successfully demonstrate, for the first time, a decoder design with a 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 article:

1) a distributed algorithm that implements the UF decoder that can exploit parallel computing units to stop decoding time per measurement round from growing with the code distance d;
2) the Helios architecture and its FPGA-based implementation that realize the distributed UF decoder;
3) a set of empirical data based on the FPGA implementation that demonstrates decreasing decoding time per round as d grows up to $d = 21$ on a VCU129 FPGA;
4) a set of empirical data that demonstrate that Helios can trade off resource usage for latency by decoding $d = 51$ on the VCU129 FPGA.

Helios is open source and available from [17].

This work contains development to the previous work by modifying Helios to support circuit-level noise, erasure errors, sliding window decoding, and tradeoff latency for lower resource usage using context switching.

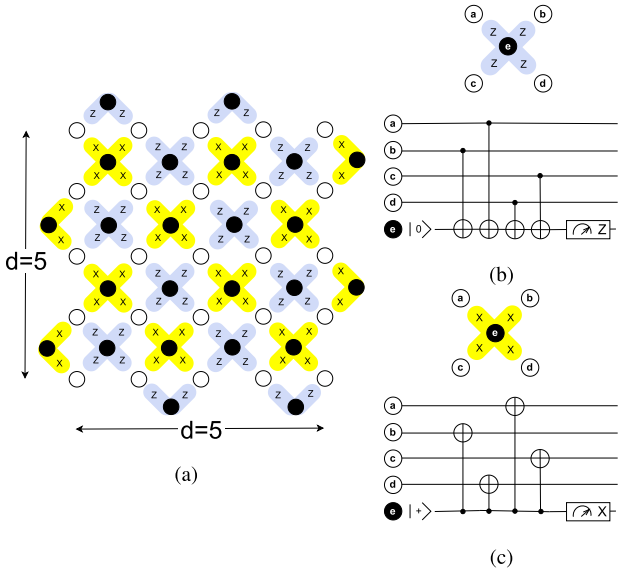

FIGURE 1. (a) Rotated CSS surface code ($d = 5$), a commonly used type of surface code. The white circles are data qubits and the black are the Z-type and X-type ancillas. (b) and (c) Measurement circuit of Z-type and X-type ancillas. Excluding the ancillas in the border, each Z-type and X-type ancilla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

II. BACKGROUND

A. ERROR CORRECTION AND SURFACE CODE

QEC is more challenging than classical error correction due to the nature of quantum bits (qubits). First, qubits cannot be copied to achieve redundancy due to the no-cloning theorem. Second, the values of 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 Fig. 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 potentially 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.

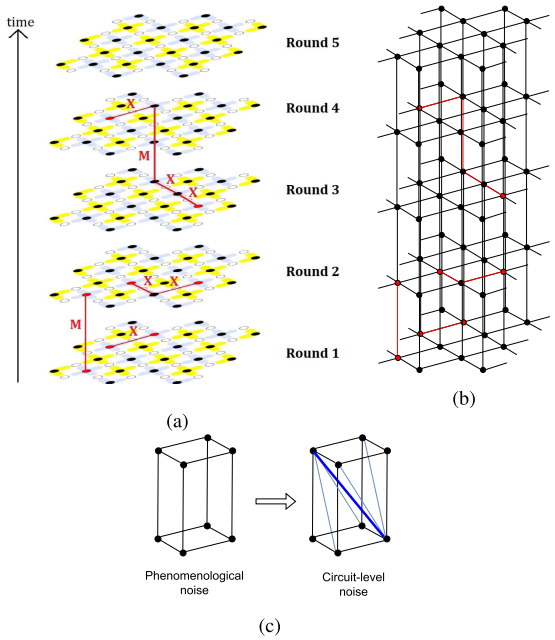

FIGURE 2. (a) Example syndrome of Z stabilizers for $d = 5$ surface code with five rounds of measurements. The syndrome contains an isolated X error (round 1), an isolated measurement error (rounds 1 and 2), a chain of two X errors (round 3), and a chain containing X errors and measurement errors spanning multiple measurement rounds (rounds 3 and 4). (b) Phenomenological noise decoding graph with defect vertices marked red for the syndrome in (a). (c) Modification of decoding graph from phenomenological noise to circuit level shown only for eight adjacent vertices. Extra edges in the circuit-level noise decoding graph are shown in blue. The thick blue edge represents a hook error and others represent X errors spanning two measurement rounds.

Noise model: A noise model defines the types and locations of X and Z errors in a surface code. The two prevalent models are the phenomenological noise model and the circuit-level noise model. In the phenomenological model, errors are confined to data qubits and occur before gate execution, consequently flipping adjacent ancilla qubits in the same measurement round. Conversely, in the circuit-level model, errors can arise between gates, resulting in the flipping of only one adjacent ancilla qubit in the current measurement round and the other in the subsequent round. In addition, in the circuit-level noise model, errors may occur in ancilla qubits, which effectively equates to errors in the two adjacent data qubits. Such errors are termed *hook errors*.

Besides these errors, there are measurement errors and erasure errors. A measurement error occurs when an ancilla qubit produces an erroneous reading due to faults in the readout process. An erasure error represents a detectable leakage of the quantum state in a qubit.

A measurement outcome of a flipped ancilla qubit is called a *defect measurement*. The outcomes from multiple rounds of measurements of ancilla qubits constitute a *syndrome*. In practice, a syndrome consists of at least d measurement rounds. Fig. 2(a) 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 each vertex represents a measurement outcome of an ancilla. An edge in this graph corresponds to an independent error source, linking two vertices that represent the defective measurement outcomes caused by this error. Thus, the number of edges in the graph depends on the error model under consideration, and the weight of an edge is determined by the probability of the error corresponding to the edge.

The decoding graph typically contains $(d + 1) \times ((d - 1)/2) \times d$ vertices. Fig. 2(b) illustrates the decoding graphs for the phenomenological noise model, and Fig. 2(c) illustrates how it will be extended for the circuit-level noise model. Notably, in the phenomenological noise model, each vertex has a maximum of six incident edges. Conversely, in the circuit-level noise model, a vertex can have up to 12 incident edges.

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.

1) METRICS

The two important aspects of decoders are accuracy and speed. A decoder must correct errors faster than syndromes that are produced to avoid a backlog. A faster decoder allows faster execution of a quantum computer, reducing the idle time waiting for decoding to be available. 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.

2) UF DECODER

The UF decoder is a fast surface code decoder design first described by Delfosse and Nickerson [15]. According to [19], it can be viewed as an approximation to the blossom algorithm that solves 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

Algorithm 1: UF 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
```

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–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 are no more odd clusters, the decoder finds a correction within each cluster and combines them to produce the correction pattern (L25).

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 [19], [20]. 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 the PE and vertex in an interexchangeable 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 *cid*s, 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 Section 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.

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
```

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

1) v, the vertex it is associated with;
2) $v.E$, the set of edges incident to v;
3) $v.U$, 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 [15]. For each vertex $v \in V$, the following information is added.

1) *id* is a unique identity number which ranges from 1 to n where $n = |V|$. *id* is statically assigned and never changes.
2) 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.
3) *cid* is 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.
4) *odd* is a binary state indicating whether the cluster is odd. *odd* is initialized to be m.
5) *codd* is a copy of *odd*.
6) `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.
7) *st_odd* is a binary state representing the parity of m of v and all its descendants.
8) `stage` indicates the stage the PE currently operates in
9) `busy` is a binary state indicating whether the PE has any pending operations.

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

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
```

For clarity of exposition, we introduce a mathematical shorthand v.`nb`, the set of vertices connected with v by full-grown edges, i.e., v.`nb`$=\{u | e = \langle v, u\rangle \in v.E \wedge e$.`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.`child` to indicate all child vertices of a vertex in the tree representation, i.e., v.`child`$=\{u | u$.`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 that both read and write are atomic. Fig. 4 illustrates these memory blocks.

Algorithm 6: 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
```

1) 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).
2) 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).
3) Memory for $e.\texttt{growth}$, which can be written by its two incident PEs (S3).
4) 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 Section 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) 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.

1) 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.

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 $v.\texttt{busy}$ as `false`. In the Checking stage, the 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 $v.\texttt{stage}$ 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. WORST CASE TIME COMPLEXITY ANALYSIS

The worst case time complexity of our distributed UF decoder is no worse than $O(d^4 \log(d))$, which is the product of the worst case number of stages, $O(d^4)$, and the worst case time complexity of the controller to change stages, $O(\log(d))$.

We show the worst case number of stages is no worse than $O(d^4)$ as follows. The number of stages is bounded by the maximum number of *Merging* and *Checking* stages (L82–L89) per iteration of the `while` loop in L77, times the number of iterations. These stages in each iteration implement a shared memory based flooding and convergecast algorithm

for all existing clusters in parallel [21]. This algorithm has a worst case time complexity of $O(d^3)$, where d^3 bounds the cluster size [21]. Because each stage implements a step in the flooding and convergecast algorithm, the maximum number of stages in each iteration is bounded by $O(d^3)$.

The number of iterations is bounded by d as each iteration consists of a *Growing* stage and the maximum number of iterations any cluster can grow is d. Thus, the total number of stages is no worse than $O(d^4)$.

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 $O(\log(d))$.

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 Section 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 Fig. 3 and explained as follows.

1) 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.
2) Shared memory S3 (e.growth) is added to the incident PE with the lower id.
3) 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 Fig. 3 represent these links.
4) The controller is realized as a tree of control nodes (see Section IV-B). The leaf nodes of the tree contain shared memory S4.
5) 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 Fig. 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.

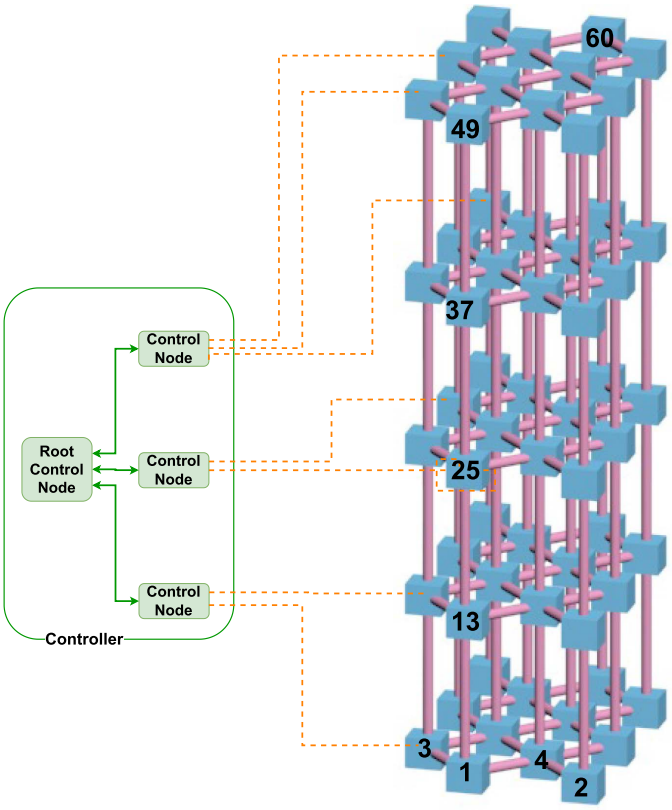

FIGURE 3. Helios architecture for $d = 5$ surface code for five measurement rounds for phenomenological noise model. As $d = 5$ surface code has 12 ancilla qubits of Z-type, Helios contains a 12 × 5 PE array. PE n indicates PE with $v.id = n$. Not all links from the controller to PEs and all $v.id$s are shown in the figure. The architecture for circuit-level noise has additional links between PEs corresponding to the additional edges in the decoding graph of circuit-level noise.

V. FPGA IMPLEMENTATION

We next describe an implementation of Helios targeting a single FPGA. We choose an FPGA for two reasons. It supports massively parallel logic, which is essential as the number of PEs grows by 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 [17].

A. LEVERAGING GLOBAL SYNCHRONIZATION IN THE 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. First, we run the Merging (L121) and Checking stages (L139) in parallel within each PE. The tight synchronization of all PEs guarantees that false negative `busy` signals do not occur.

Second, 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).

In addition, 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`.

Algorithms 7 and 8 lists the FPGA-oriented algorithm of PE and the controller. The logic at every positive edge is executed in parallel. Fig. 5 provides a simple example of how the FPGA implementation merges a cluster of four defect measurements in eight cycles.

1) TIME COMPLEXITY

The worst case cycle count of the FPGA design is bounded by $3d^4 + 2d$. The merging stage consists of three primary operations: a broadcast (cid), a convergecast (st_odd), and another broadcast (odd) with each operation requiring at most d^3 cycles. In addition, the merging stage needs an extra cycle to verify completion. Conversely, the growth stage requires a single cycle. As a result, each iteration requires at most $3d^3 + 2$ cycles. Since the number of iterations is at most d, the worst case cycle count is bounded by $3d^4 + 2d$. The worst case time complexity of the FPGA design is $O(d^4)$ in contrast to $O(d^4 \log(d))$ of the distributed UF algorithm. The $\log(d)$ factor in the latter originates from the coordination overhead associated with transitioning between the stages. The FPGA design performs stage transition in a single cycle, effectively eliminating the $\log(d)$ factor.

B. OPTIMIZING RESOURCE EFFICIENCY

As the resource usage grows $O(d^3 \log(d))$, the number of lookup tables (LUTs) limits the largest d that can be implemented on a given FPGA. To address this constraint, we adopt a method first proposed by Heer et al. [22]. This method first partitions the decoding graph into multiple subgraphs and then time-multiplexes them in the FPGA.

We first partition the decoding graph into multiple subgraphs by splitting it evenly along one or more axes. This even partitioning ensures that each subgraph is roughly the same size, thereby increasing resource utilization. The necessary condition for partitioning is that each subgraph must be sized to fit within a single FPGA.

Time multiplexing of multiple subgraphs is as follows. We first implement a graph in the FPGA with the same topology as a subgraph and at least as large as the largest subgraph, which we will call a *lattice*. We then iteratively map each subgraph to the lattice during each decoding stage. All subgraphs of the decoding graph can be mapped to the same lattice due to the homogeneous topology of the decoding graph where each PE has a fixed number of edges that connect to adjacent PEs. If n subgraphs timeshares a lattice, we denote the implementation as Helios-n. By default, Helios denotes Helios-1 when there is no multiplexing.

We implement context switching between subgraphs at the PE level. We augment each PE in the lattice, which we label as a *physical PE*, with a local memory. During context switching, each PE stores its PE states in the local memory and loads the PE states of the corresponding PE of the next subgraph from the local memory. This is akin to context switching of threads in an operating system. Fig. 4 shows a minimal diagram of a physical PE. In the FPGA, this local memory is mapped to LUTRAM rather than block RAM (BRAM), due to its shallower depth. We also note that context switching in Helios consumes a single cycle as it is essentially reading and writing from local memory.

In the example in Fig. 5, adding context switching requires two additional cycles for the cluster, occurring after the growing stage (cycle 1) and the merging stage (cycle 8).

A careful reader may point out that time multiplexing through multiple subgraphs would require extra connections between physical PEs to provide adjacent PE information to virtual PEs mapped to the boundary of the lattice. Indeed, this is the case for the original design proposed by Heer et al. [22]. We avoid these extra connections by carefully mapping virtual PEs to physical PEs. We always map any pair of adjacent virtual PEs in the decoding graph belonging to different subgraphs, to the same physical PE. Thus, the missing information of adjacent PEs at the lattice boundary can be loaded from the local memory of the physical PE.

C. IMPLEMENTATION DETAILS

We next list the other implementation choices of our design.

1) CONTEXT SWITCHING

On the VCU129 FPGA development board [16], without context switching, we can support the distributed UF decoder with d up to 21 for the phenomenological noise model and up to 17 for circuit-level noise, due to resource limits. We use context switching only for d exceeding those limits. Furthermore, we restrict the partitioning of the decoding graph along a single axis to avoid excessive sequential reads from local memory by physical PEs at the boundary of the lattice.

2) CONTROLLER

Since the largest number of PEs we can implement a single VCU129 FPGA is 4620 ($d = 21$), a single-node controller

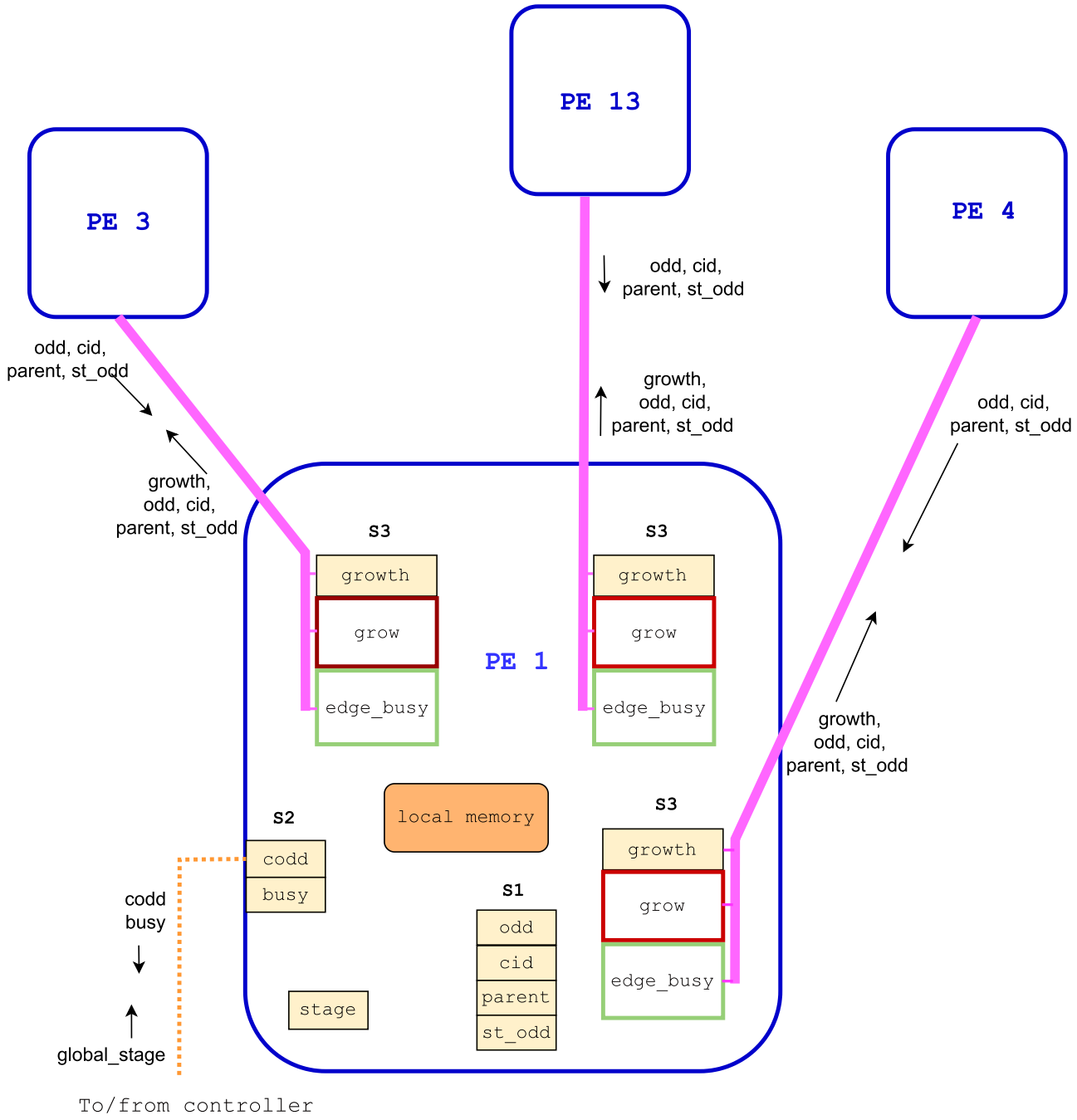

FIGURE 4. Bottom left corner of the PE array shown in Fig. 3. Only part of the logic and memory inside PE 1 is shown: `growth` **(S3) is per edge and is stored in the PE with lower *id*.** `grow` **logic (in brown) calculates the updated growth value.** `edge_busy` **(in green) is per adjacent PE and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

suffices. The node controller reads `busy` of each PE, every clock cycle to identify the completion of a stage.

3) 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 Fig. 6. 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.

D. RESOURCE USAGE

Table 1 shows the resource usage for various d for phenomenological noise model and circuit-level noise.

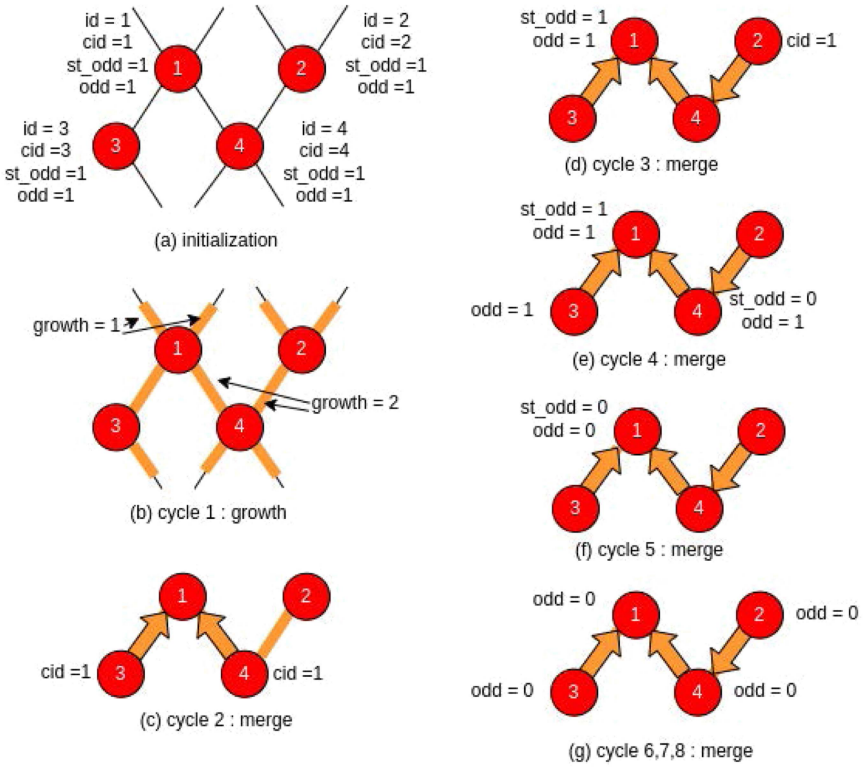

FIGURE 5. Example figure showing how the FPGA implementation groups four nearby defect measurements into a single cluster in eight cycles. (a) Each defect measurement is mapped to a PE, and initially, the four defect measurements have $v.id = 1, 2, 3, 4$, $v.cid = v.id$, and $v.st\_odd = v.odd = 1$. (b) First growth cycle results in fully grown edges between {1,3}, {1,4}, and {2,4}. (c) During merging, PEs 3 and 4 set their $v.cid$ as 1 and set their parents to 1 (shown with orange arrows). (d) In the next cycle, PE 1 calculates the parity of the subtree rooted at 1 (PEs 1, 3, 4), while PE 2 updates its $v.cid$ and parent. (e) and (f) This results in an update of $v.st\_odd$ of subtrees rooted at 4 and 1 in the next two cycles. Simultaneously, the root node (PE 1) updates the parity of the cluster ($v.odd = 0$). (g) $v.odd$ is propagated to all PEs in the cluster in two cycles, and no change occurring in the eighth cycle tells the controller to advance the stage.

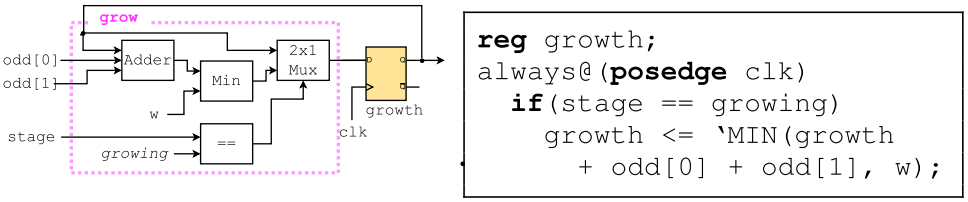

FIGURE 6. Circuit diagram of `grow` submodule and Verilog implementation. This implements the atomic compare and update operation in L45 as part of the PE module. *odd*[0] and *odd*[1] represent the *odd* states of the two incident PEs of the edge.

TABLE 1. Resource Usage of Helios on VCU129 FPGA Board for Selected *d*

d	phenomenological			circuit-level		
	# LUTs	# regs	$f_{\max}$	# LUTs	# regs	$f_{\max}$
3	970	528	260	1557	699	195
5	6425	2425	220	11128	3469	130
9	52111	13754	175	93797	22515	110
13	165718	47211	140	340084	74927	100
17	448314	122028	125	888854	177284	75
21	898715	238939	100	n/a	n/a	n/a

LUTs and # Regs show the number of LUTs and registers required for each configuration, and f_{max} shows, in MHz, the maximum clock frequency each configuration can run.

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 3-D array, as shown in Fig. 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. The increased number of incident edges also causes the Helios for circuit-level noise to use twice the resources as in circuit-level noise each PE inside the graph can have up to 12 incident edges.

Existing commercial FPGAs like VCU129 often dedicate a lot of silicon to digital signal processing (DSP) units and BRAMs. However, our design does not use any DSPs because it only requires comparison operators and fixed point

additions. We only use BRAMs to support interfacing with the MicroBlaze core and implement context switching using LUTRAMs instead of BRAMs 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.

E. CLOCK FREQUENCY

The architectural mismatch between the 3-D design of Helios and the 2-D structure of an FPGA creates a fundamental limitation in the maximum clock frequency Helios can run when implemented on an FPGA. Despite Helios's capability to scale to arbitrarily large d, the maximum clock frequency of the FPGA implementation must decrease as d increases and eventually Helios will not be able to decode at the rate of measurement. In our implementation, we reach the limit of FPGA resources before the clock frequency becomes the bottleneck, at $d \leq 51$. We estimate that with an arbitrarily large FPGA with the same routing technology as Virtex UltraScale+ device, Helios's clock frequency will become the bottleneck and will fail to decode at the rate of measurement at around $d \approx 1800$.

The signal propagation latency increases by $O(d)$ in the FPGA implementation, causing a decrease in maximum clock frequency as observed in Table 1. When d increases, PEs adjacent in Helios must be placed farther apart within the FPGA, causing this increase. Specifically, the critical path's routing latency increases from 3.77 ns for a $d = 3$ circuit-level noise model design to 11.56 ns for a $d = 17$ design. In addition, the increase in bit width of d increases the logical processing latency by $O(\log(\log(d)))$, which is significantly less compared to the delay due to propagation. When we synthesize PEs in isolation, the logic delay increases slightly from 1.475 to 1.552 ns when increasing d from 5 to 21. We should also note that the significantly high maximum operating frequency for $d = 3$ under phenomenological noise is due to the unique situation of the decoding graph for $d = 3$, where no PE has more than three incident edges.

A potential approach to circumvent this architectural mismatch is to preserve Helios's 3-D structure by mapping the decoder across multiple FPGAs. However, the limitation of I/O pins in existing FPGAs and significant inter-FPGA latency of a few tens of nanoseconds prohibit practical implementation of Helios across multiple FPGAs efficiently compared to a single FPGA implementation.

Implementation choice: For most experiments, we synthesize the design targeting a clock frequency of 100 MHz. This choice ensures sufficient latency for completing the critical path within a single clock cycle, allowing for a uniform comparison of the effects of our distributed UF decoder. We used slower clock frequencies, which were necessary due to resource congestion, only for the implementations of circuit-level noise at $d = 17$ and the resource-efficient implementation at $d = 51$.

F. POWER CONSUMPTION

The power consumption of the implementation depends upon the number of PEs actively participating in the clustering process. For $d = 13$, the Vivado synthesizer estimates power consumption at 4.639 W for the FPGA implementation. This estimation is based on assuming random input values toggled continuously, which results in all PEs being active at the same time [23]. The power consumption during decoding is likely to be much lower because most syndromes contain only a small number of defect measurements, and as a result, only a small number of PEs are typically active at a time during the decoding process.

VI. EVALUATION

The main objective of our evaluation is to assess the scalability of our distributed UF implementation. To that end, we answer the following questions in our evaluation.

1) *Latency growth*: Does the latency of distributed-UF decoder grow sublinearly for both phenomenological noise and circuit-level noise?
2) *Context switching overhead*: Can we use context switching to decode large surface codes without excessive latency growth?
3) *Extensibility*: Can Helios architecture be extended to support erasure errors, weighted edges, and sliding-window decoding?

We first describe our methodology and follow that with the evaluation results to answer the aforementioned questions.

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 those of the original UF decoder. We compare clusters because the original UF decoder and ours only differ in implementing clustering. 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 as our decoder and the original UF decoder by Delfosse and Nickerson [15] produces the same decoding output for any given syndrome. Nevertheless, for completeness, Fig. 7 compares the logical error rates between our distributed UF decoder and the MWPM decoder. We obtain results in Fig. 7 using a software implementation of the UF decoder and the MWPM decoder under circuit-level noise, with each data point representing the average of 10^8 trials [24].

1) EXPERIMENTAL SETUP

As our evaluation setup, we use the Xilinx VCU129 FPGA development board [16], which contains one of the largest FPGAs available on a Xilinx development board. We simulate a surface code on a personal computer under various noise models to generate syndromes, storing the output in a file. Subsequently, a MicroBlaze soft processor core [25], instantiated within the FPGA, reads this syndrome file. The

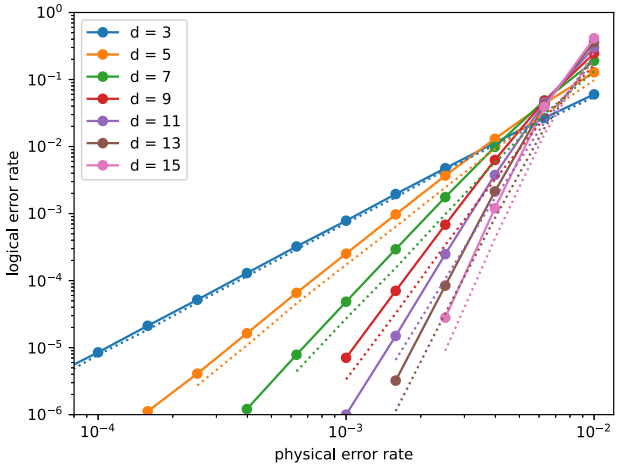

FIGURE 7. Logical error rate of distributed UF decoder (dark lines) in comparison with the MWPM decoder (dashed lines).

core then transmits the syndromes to Helios, which operates within the same FPGA. We ran 10^6 trials for each error rate and distance.

2) NOISE MODEL

We use phenomenological noise model [1], circuit-level noise model [26], and phenomenological noise model with erasure errors [15]. As decoding for X errors and Z errors are independent and identical, we only focus on decoding X errors in the evaluation.

We use three noise models in our experiments: the phenomenological noise model [1], the circuit-level noise model [24], [26], and the phenomenological noise model with erasure errors [15]. Each of these models additionally includes measurement errors. As the decoding for X errors and Z errors are independent and identical, we focus solely on decoding X errors in our evaluation.

To simulate noise, we independently flip data qubits and ancilla qubits in our simulation model. In the phenomenological noise model, data qubits are independently flipped between each measurement round with a probability p. For circuit-level noise, we flip both data and ancilla qubits between each pair of gates and between gates and measurements, also with a probability p. For erasure errors, we erase data qubits between measurement rounds with a probability p_e, and ancilla qubits adjacent to the erased qubit are flipped with a 50% chance to emulate erasure effects. To emulate measurement errors, we flip ancilla qubits with a probability of p. This is a widely used approach by prior QEC decoders [7], [11], [14], [15], [26], [27]. 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], [10], [14]. 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 $p = 0.024$ for phenomenological noise calculated by Delfosse and Nickerson [15]. Similarly, for circuit-level noise, the UF decoder has a threshold of $p = 0.0078$ calculated by Barber et al. [10].

B. DECODING TIME

We experimentally show how the average decoding time grows with the surface code size for different noise models.

1) AVERAGE TIME

To demonstrate the scalability of our algorithm with respect to the size of the surface code, we measure the average time for decoding for various sizes of the surface code. Fig. 8(a) shows the average decoding time in nanoseconds grows sublinearly with the distance (d) of the surface code (x-axis). We see that for both phenomenological noise and circuit-level noise we tested against, average decoding time grows sublinearly 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 Fig. 8(b).

2) DISTRIBUTION OF DECODING TIME

To understand the growth of decoding time with respect to the code distance, in Fig. 9(a), 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. We indicate the average cycle count 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 Fig. 9(a) 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 Fig. 9(a), the distribution for each surface code size is right-skewed. For $d = 13$, as seen in Fig. 10, 97% of trials required two iterations or fewer, which were completed within 250 ns. In the same test, 99.99% of trials were completed within 510 ns. Only exponentially fewer error patterns require long decoding times, corresponding to syndromes with longer error chains, which contributes less to the average decoding time. For example, excluding the 0.01% samples in the tail yields an average decoding time of 194.16 ns, compared to 194.19 ns for all samples.

The longest decoding latency we observe in our trials, 920 ns, is significantly lower than the theoretical worse case

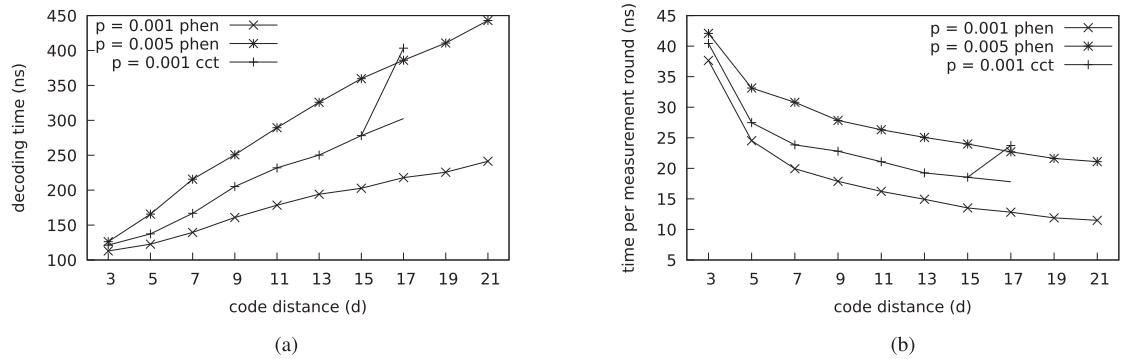

FIGURE 8. Average decoding time scales sublinearly with d. We measure the average decoding time for phenomenological noise (*phen*) of 0.005 and 0.001 and circuit-level noise (*cct*) of 0.001. (Left) Average decoding time. The average time per measurement round reducing continuously justifies that our decoder is scalable for large surface codes under both phenomenological noise and circuit-level noise. The unusual increase at $d = 17$ for circuit-level noise is caused by reducing the operating frequency to 75 MHz. The dashed line shows the calculated value at 100 MHz. We show the distributions separately in Fig. 9(a). (a) Average decoding time. (b) Average decoding time per measurement round.

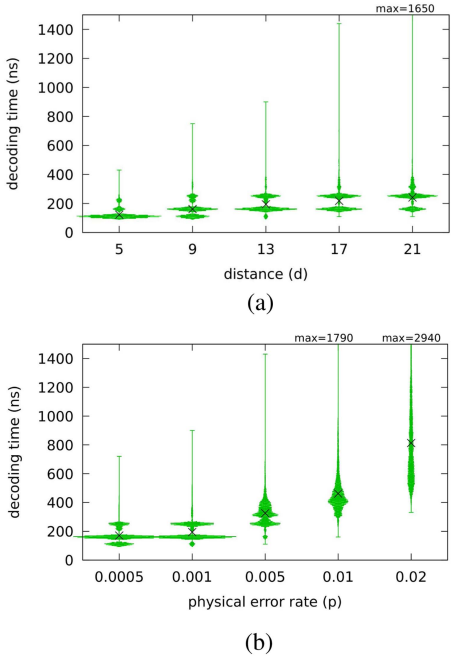

FIGURE 9. Distribution of decoding time (T) with the mean marked with ×. Each distribution includes 10^6 data points. By default, $d = 13$, and phenomenological noise of $p = 0.001$ is unweighted. (a) T increases with d. (b) T grows with the physical error rate.

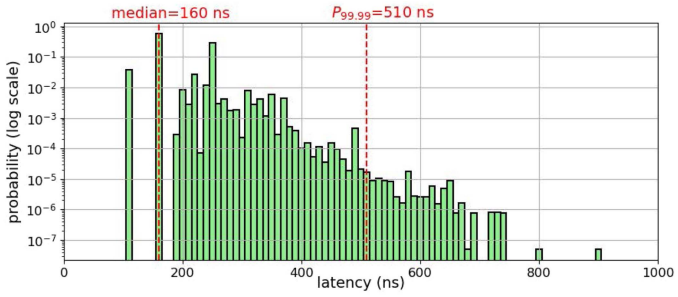

FIGURE 10. Histogram of decoding time (latency) from 2×10^7 data points at $d = 13$ and $p = 0.001$ phenomenological noise. This shows long decoding times are exponentially unlikely.

decoding time of 857 μs, calculated by the equation in Section V-A. This discrepancy arises because the worst case scenario requires a very specific pattern of syndromes, which is exceedingly rare and highly unlikely to occur in typical simulation settings.

3) EFFECT OF PHYSICAL ERROR RATE

To understand the effect of the physical error rate on decoding time, in Fig. 9(b), we plot the distribution of latency for five different noise levels for $d = 13$. The y-axis shows the latency and the x-axis 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. For the highest tested physical error rate of 0.02, the average decoding time is 814.6 ns. Thus, even when the physical error rate is closer to the threshold, the decoder is an order of magnitude faster than the rate of measurement.

C. EFFECT OF OPTIMIZING FOR RESOURCE USAGE

We next show that Helios can decode surface codes larger than $d = 21$ by dividing and time multiplexing the decoding graph. We first show that Helios can decode $d = 27$, a possible d to run useful quantum algorithms [5], and then show Helios can even decode significantly large d such as 51.

In Fig. 11(a), we plot the average latency for decoding $d = 27$ surface code under phenomenological noise of 0.001 and 0.005 for Helios-4 to Helios-27. The Y-axis shows the average latency and the X-axis shows the number of LUTs required for implementation. We only show the average as it is the critical factor enabling backlog-free decoding and the distribution observes a similar pattern as distributions shown in Fig. 11(c). In Fig. 11(b), we plot the corresponding LUT count for Helios-n configurations shown in Fig. 11(a). We use the LUT count to indicate resource usage because it is the

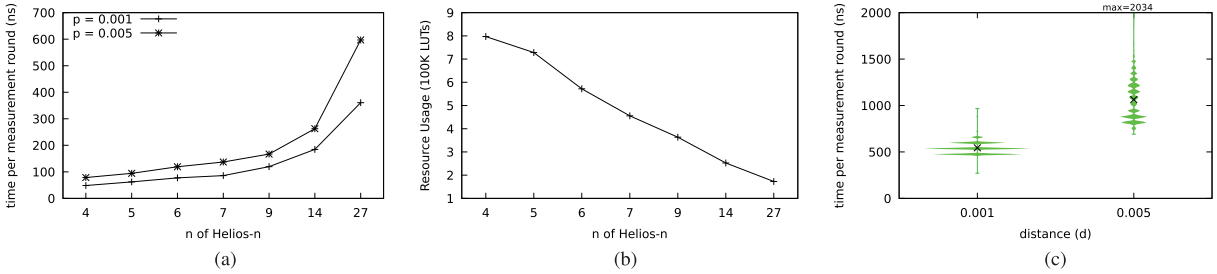

FIGURE 11. Helios can optimize for resource usage by mapping multiple virtual PEs to a single physical PE. (a) Average latency per measurement round for $d = 27$ under two different phenomenological noise levels. (b) Corresponding resource use for the Helios-n configurations for $d = 27$. (c) Distribution of decoding time for $d = 51$ with Helios-51. This configuration can decode faster than the rate of measurement for $p = 0.001$, but is slightly slower than the rate of measurement for $p = 0.005$. (a) Latency increases with n. (b) Resource use decreases with n. (c) Helios can decode $d = 51$.

limiting factor when running a decoder on a given FPGA. In Fig. 11(a) and (b), we select Helios-n configurations resulting in maximum resource utilization. Due to the restriction of partitioning solely across the measurement round axis, certain mappings, like Helios-8, are inefficient. Helios-8 will have a lattice with a height of four physical PEs, but the PEs in the topmost layer would only have three virtual PEs mapped to each physical PE. Conversely, Helios-7 results in the same number of physical PEs but with a lesser number of context switchings.

Fig. 11(a) shows that under phenomenological noise of 0.001, Helios can decode a $d = 27$ surface code at an average latency of 48.5 ns per measurement round by mapping four virtual PEs to each physical PE. This rate is over 20 times faster than the measurement rate. By mapping 27 virtual PEs to each physical PE, resource usage can be further reduced to 173K LUTs, while maintaining the ability to decode at 360 ns per measurement round. In this configuration, each physical PE cycles through a single measurement result of the corresponding ancilla in each context. The reduced LUT count is particularly significant as it enables the implementation to be mapped onto more cost-effective FPGA models.

Helios can decode $d = 51$ faster than the rate of measurement for phenomenological noise of $p = 0.001$ but is slightly slower than the rate of measurement when $p = 0.005$. The average latencies for the noise level above are 543.9 and 1064.0 ns, respectively. This design targeting $d = 51$ required around 796K LUTs and operates at 85 MHz. Increased resource utilization at $d = 51$ causes the reduction in operating frequency. The distribution of latency is shown in Fig. 11(c).

D. DECODER EXTENSIONS

We next analyze the impact of extending our decoder for other requirements. We consider three situations: nonidentically distributed errors, erasure errors, and indefinite preserving of logical state.

1) NONIDENTICALLY 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.

Fig. 12(a) 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 nonidentically distributed errors can be performed in real time using distributed UF on Helios.

2) ERASURE ERRORS

The introduction of erasure errors slightly increases the decoding latency. Fig. 12(c) shows the distribution of latency when erasure errors are added on top of p. An erasure rate of 0.001 results in an increase in average decoding time by approximately 63 ns. Notably, 40 ns of this increase (four FPGA clock cycles) comes from an extra `merging` stage prior to the initial `growing` stage. This extra `merging` stage is necessary because erasure errors reduce an edge's weight to zero when an erasure occurs, which can lead to the merging of vertices before any growing of clusters. Thus, the overall latency distribution is right-shifted by 40 ns with a slight additional increase of latency due to X errors caused by erasures.

3) PRESERVING LOGICAL STATE INDEFINITELY

We next show that Helios can be extended to preserve logical state indefinitely using the sliding window method [1]. While standard decoder evaluations focus on decoding d rounds,

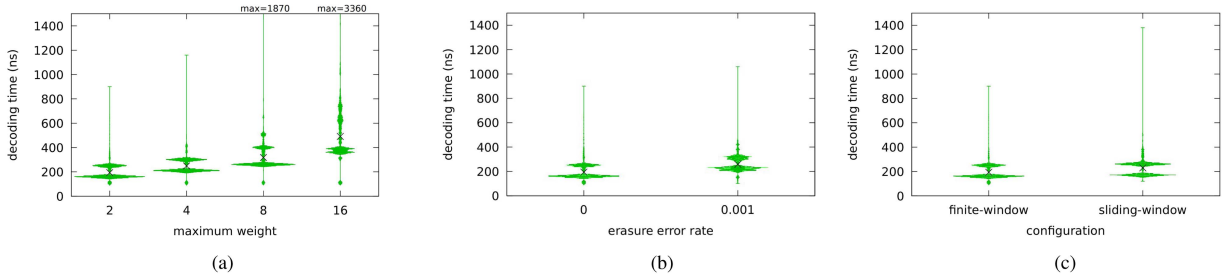

FIGURE 12. Distribution of decoding time (T) for decoder extensions. The mean is marked with ×. Each distribution includes 10^6 data points. By default, $d = 13$, and phenomenological noise of $p = 0.001$ is unweighted. (a) T grows with the weight of the edges. (b) T shifts with erasures. (c) T increases with sliding window.

TABLE 2. Implementations of Surface Code Decoders on Classical Hardware (FPGA)

Name	Algorithm	Max.		Decoding Time (ns)		Features (✓ = implemented, ∗ = easily extendable)		
		d	m	Average	Per Round	Erasures	Weighted edges	Circuit noise
LILLIPUT [6]	Lookup table	5	2	42	21.0		✓	∗
Overwater et al. [29]	Neural Network	5	1	88	87.6	∗	✓	∗
WIT-Greedy [13]	Greedy	11	d	370	33.6		✓	∗
Astrea-G [9]	Greedy	9	d	450	50.0		✓	∗
Collision Clustering [10]	Union-Find	21	d	2000	160.0			✓
Helios (This work)	Union-Find	51	d	250	19.3	✓	✓	✓

Average decoding time is provided for $d = 13$ for decoders capable of decoding $d \geq 13$ and for maximum d for other decoders. Max d is the maximum d the decoder can decode faster than the rate of measurement within the device resource budget. Max m is the maximum number of measurement rounds considered for decoding.

practical applications require a decoder to continuously process incoming measurement rounds to maintain the logical state indefinitely. The prevalent method for achieving this is the sliding window method. In this method, $2d$ rounds are decoded simultaneously, but corrections are only committed for the oldest d rounds. Subsequently, the decoding window advances by d rounds, resulting in continuous decoding for an indefinite period.

We implement sliding window decoding by extending the PE array for $2d$ measurement rounds. This results in a slight increase in latency and more than a doubling of resource usage. For instance, a $d = 13$ decoder supporting the sliding window method requires 371K LUTs, compared to the 166K LUTs needed for the finite-window version. However, due to the vertex-level parallelism, the decoding latency has a modest increase from 194 to 230 ns. Thus, even with sliding window decoding, the decoder is 56 times faster than the rate of measurement.

E. QUANTITATIVE COMPARISON WITH RELATED WORK

Our empirical results, as shown in Fig. 9(a), suggest that Helios has a lower asymptotic complexity than any existing MWPM or UF implementation for which asymptotic complexities are available, e.g., [15], [28]. Indeed, the empirical results suggest that our decoder has a sublinear 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.

In Table 2, we compare our decoder with other hardware decoders in the literature that provide implementation-based results. We report the average decoding time for $d = 13$ for decoders capable of decoding $d > 13$. For other decoders, we report the average decoding time for the maximum d it can support.

As seen in Table 2, the most notable prior implementation is the Collision Clustering decoder by Riverlane [10]. This decoder, like our work, is an alternative implementation of the UF algorithm. Its novel approach involves each vertex tracking its growth and using a hardware-implemented function for efficient distance computation between vertices to decide which vertices should merge. This reduces the memory access requirements for UF implementation, compared to similar prior designs like AFS [7]. The reuse of the distance calculation function in all merging operations results in substantially lower resource consumption for the Collision Clustering decoder compared to Helios. For example, for $d = 13$, the Collision Clustering decoder requires about 6K LUTs, whereas Helios requires 340K LUTs.

However, the average case latency per measurement round in the Collision Clustering decoder increases linearly with d, creating an upper bound of maximum d that can be decoded in real time. In contrast, Helios can decode arbitrarily large d, as shown in this work. Furthermore, the speed of the decoder relies on the efficient calculation of the distance between any two nodes. While this is straightforward for unweighted

edges, when weighted edges or erasure errors are present, the decoder requires a complex function such as Dijkstra's algorithm to calculate distances. Using such a function for distance calculation can significantly increase the decoding time, making decoding erasure errors or weighted edges faster than the rate of measurement likely prohibitive.

LILLIPUT [6], Astrea-G [9], and WIT-Greedy [13] are not scalable for large d, due to their excessively high storage requirements. LILLIPUT [6] is an LUT-based decoder. LUT-based decoders can achieve fast decoding but are not scalable beyond $d = 5$ as the LUT size grows $O(2^{d^3})$. For $d = 7$ surface code with seven measurement rounds, it would require a memory of 2^{168} bytes, which is infeasible in any foreseeable future. Astrea-G [9] and WIT-Greedy [13] store weights of all pairs of vertices and compare probable matchings. Astrea-G uses a greedy algorithm to preselect matchings, and WIT-Greedy selects the least weight matching directly using a greedy algorithm, reducing accuracy further. The memory requirement for their weight tables grows $O(d^6)$, limiting their implementations at $d = 9$ and $d = 11$, respectively. In contrast, our work has successfully demonstrated the implementation of a $d = 51$ surface code on a VCU129 FPGA. Furthermore, while these decoders could be adapted to circuit-level noise, accommodating erasure errors would exacerbate their already substantial memory requirements, due to the need to process additional erasure inputs.

Overwater et al. [29] implement a neural-network-based decoder. As shown by the authors, the decoder requires 44K LUTs for distance 5 for a single measurement round. This worsens with distance as the input layer scales $O(d^2)$ with distance and $O(d^3)$ if d rounds of measurements are considered. In comparison, Helios with $d = 5$ with five measurement rounds requires only 11K LUTs.

Our decoder outperforms the two fastest software MWPM decoder, Sparse Blossom [8] and Fusion Blossom [14], by an order of magnitude. According to our evaluation, Sparse Blossom and Fusion Blossom take 160 and 295 ns per measurement round, respectively, for $d = 13$ under $p = 0.1\%$ phenomenological noise, 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., [30], [31], [32], [33]. The most related are solutions that leverage parallel computing resources.

Fowler [28] 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 interunit 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+ [11] and QECOOL [12] parallelize computation at the ancilla level, where a single compute unit handles all vertices in the decoding graph representing measurements of one ancilla. This results in an increase in decoding time per measurement round as d increases. In contrast, we allocate a PE per 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, which is much lower in accuracy than the UF decoder used in this work. QECOOL has an accuracy of approximately four orders of magnitude lower than a UF decoder [7], and NISQ+ ignores measurement errors, further lowering its accuracy than QECOOL.

Wu and Zhong [14], Skoric et al. [27], and Tan et al. [34] 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 faster decoding rate than the measurement rate. 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.

Since the initial release of this work [18], two alternative designs employing vertex-level parallelism have been reported: Actis [35] and Heer et al. [36]. Both map each vertex to a PE and support nearest neighbor communication. However, unlike our approach, these designs incorporate communication of PEs with the central controller through the vertex array, resulting in a notable increase in coordinating overhead. Furthermore, no implementation has been reported for either of them, making a direct comparison in terms of resource usage difficult.

Pipelining can be considered a special form of using compute resources in parallel, i.e., in different pipeline stages. Examples include AFS [7], LILLIPUT [6], Astrea-G [9], and Collision Clustering [10]. However, pipelining is limited in how much parallelism it can leverage: the number of pipeline stages. This results in a maximum d, which they can decode faster than the rate of measurement. The largest d reported for pipelined decoders is $d = 23$, which the application-specific integrated circuit (ASIC) design described in [10] achieves

Algorithm 7: FPGA-Oriented Algorithm for Vertex v in the Distributed UF Decoder.

```
96  v.cid ← v.id; v.odd ← v.m; v.parent ← v.id;
      v.st_odd ← v.m
97
98  % Stage transition logic
99  At every positive clock edge do
100     if global_stage =terminate then return
101     else if global_stage =growing then
          v.stage ← growing
102     else if v.stage =growing then v.stage ← merging
103 end
104
105 % Growing logic
106 At every positive clock edge do
107     if v.stage =growing then
108         for each e = ⟨u, v⟩ ∈ v.E and v.id < u.id do
109             if e.growth< e.w and u.cid ≠ v.cid then
110                 if v.odd and u.odd then
111                     e.growth← MIN(e.growth+2, w)
112                 end
113                 else if v.odd or u.odd then
114                     e.growth← MIN(e.growth+1, w)
115                 end
116             end
117         end
118     end
119 end
120
121 % Merging logic
122 At every positive clock edge do
123     Let u be arg min_{u∈(v.nb ∪ {v})}(u.cid)
124     if u.cid < v.cid then
125         v.cid ← u.cid
126         v.parent ← u.id
127     end
128 end
129
130 At every positive clock edge do
131     v.st_odd ← subtree_parity(v)
132 end
133
134 At every positive clock edge do
135     if v.parent = v.id then v.odd ← v.st_odd
136     else v.odd ← u.odd where u.id = v.parent
137 end
138
139 % Checking logic
140 At every positive clock edge do
141     if ∃u ∈ v.nb, (u.cid ≠ v.cid ‖ v.odd ≠ u.odd) then
142         v.busy ← true
143     end
144     else if v.st_odd ≠ subtree_parity(v) then
145         v.busy ← true
146     end
147     else if (v.parent = v.id & v.odd ≠ v.st_odd) then
148         v.busy ← true
149     end
150     else
151         v.busy ← false
152     end
153 end
154
155 function subtree_parity(v)
156     parity ← v.m
157     for each u ∈ v.child do
158         parity ← XOR(parity, u.st_odd)
159     end
160     return parity
161 end
```

with 240 ns per measurement round. The parallelism of our decoder grows along d^3, which enables us to achieve a sublinear average case latency, including decoding $d = 23$ within 24.1 ns. However, Helios uses significantly more resources due to increased parallelism.

VIII. CONCLUSION

In this article, we describe a distributed design for the UF 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 sublinearly 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. Furthermore, to address resource constraints, Helios can efficiently reuse FPGA resources, albeit with increased latency. We experimentally demonstrate that Helios can decode extremely large surface codes such as $d = 51$ on a VCU129 FPGA, which validates that Helios can support the surface code of any useful distance.

APPENDIX A
FPGA-ORIENTED ALGORITHM

Algorithm 8: FPGA-Oriented Controller Logic.

```
162 global_stage ← growing
163 At every positive clock edge do
164     if global_stage = growing then
165         global_stage ← merging
166         %Wait until all PEs are in Merging Stage
167         Wait 2 clock cycles
168     end
169     else if ∀v ∈ V, v.busy = false then
170         if ∀v ∈ V, v.codd = false then
171             global_stage ← terminate
172         end
173         else
174             global_stage ← growing
175         end
176     end
177 end
```

In Algorithms 7 and 8, we show the FPGA-oriented algorithm for distributed UF.

REFERENCES

[1] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *J. Math. Phys.*, vol. 43, no. 9, pp. 4452–4505, 2002, doi: 10.1063/1.1499754.

[2] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Phys. Rev. A*, vol. 86, no. 3, 2012, Art. no. 032324, doi: 10.1103/PhysRevA.86.032324.

[3] J. P. B. Ataides, D. K. Tuckett, S. D. Bartlett, S. T. Flammia, and B. J. Brown, "The XZZX surface code," *Nat. Commun.*, vol. 12, no. 1, Apr. 2021, Art. no. 2172, doi: 10.1038/s41467-021-22274-1.

[4] Z. Chen et al., "Exponential suppression of bit or phase errors with cyclic error correction," *Nature*, vol. 595, no. 7867, pp. 383–387, Jul. 2021, doi: 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, Apr. 2021, Art. no. 433, doi: 10.22331/q-2021-04-15-433.

[6] P. Das, A. Locharla, and C. Jones, "LILLIPUT: A lightweight low-latency lookup-table decoder for near-term quantum error correction," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 541–553, doi: 10.1145/3503222.3507707.

[7] P. Das et al., "AFS: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2022, pp. 259–273, doi: 10.1109/HPCA53966.2022.00027.

[8] O. Higgott and C. Gidney, "Sparse blossom: Correcting a million errors per core second with minimum-weight matching," 2023. *arXiv:2303.15933*, doi: 10.48550/arXiv.2303.15933.

[9] S. Vittal, P. Das, and M. Qureshi, "Astrea: Accurate quantum error-decoding via practical minimum-weight perfect-matching," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2023, pp. 1–16, doi: 10.1145/3579371.3589037.

[10] B. Barber et al., "A real-time, scalable, fast and highly resource efficient decoder for a quantum computer," 2023, *arXiv:2309.05558*, doi: 10.48550/arXiv.2309.05558.

[11] 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," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2020, pp. 556–569, doi: 10.1109/ISCA45697.2020.00053.

[12] 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 Des. Automat. Conf.*, pp. 451–456, 2021,doi: 10.1109/DAC18074.2021.9586326.

[13] W. Liao, Y. Suzuki, T. Tanimoto, Y. Ueno, and Y. Tokunaga, "WIT-Greedy: Hardware system design of weighted iterative greedy decoder for surface code," in *Proc. ACM Asia South Pacific Des. Automat. Conf.*, 2023, pp. 209–215, doi: 10.1145/3566097.3567933.

[14] Y. Wu and L. Zhong, "Fusion blossom: Fast MWPM decoders for QEC," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2023, pp. 928–938, doi: 10.1109/QCE57702.2023.00107.

[15] N. Delfosse and N. H. Nickerson, "Almost-linear time decoding algorithm for topological codes," *Quantum*, vol. 5, Art. no. 595, Dec. 2021, doi: 10.22331/q-2021-12-02-595.

[16] Xilinx, "Virtex UltraScale 56G PAM4 VCU129 FPGA evaluation kit." Accessed: Feb. 14, 2024. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vcu129.html

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

[18] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, "Scalable quantum error correction for surface codes using FPGA," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2023, pp. 916–927, doi: 10.1109/QCE57702.2023.00106.

[19] Y. Wu, N. Liyanage, and L. Zhong, "An interpretation of union-find decoder on weighted graphs," 2022, *arXiv:2211.03288*, doi: 10.48550/arXiv.2211.03288.

[20] S. Huang, M. Newman, and K. R. Brown, "Fault-tolerant weighted union-find decoding on the toric code," *Phys. Rev. A*, vol. 102, no. 1, Jul. 2020, Art. no. 012419, doi: 10.1103/PhysRevA.102.012419.

[21] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, 2nd ed. Hoboken, NJ, USA: Wiley, 2004.

[22] M. J. Heer, J.-E. R. Wichmann, and K. Sano, "Achieving scalable quantum error correction with union-find on systolic arrays by using multi-context processing elements," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2023, pp. 242–243, doi: 10.1109/QCE57702.2023.10224.

[23] V. Xilinx, "Design suite user guide: Power analysis and optimization, Xilinx," Oct. 2021, UG907. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2021_2/ug907-vivado-power-analysis-optimization.pdf#page=16.10

[24] "QEC Playground," 2023. [Online]. Available: https://github.com/yuewuo/QEC-Playground

[25] Xilinx, "MicroBlaze processor quick start guide." Accessed: Feb. 14, 2024. [Online]. Available: https://docs.xilinx.com/v/u/en-US/microblaze-quick-start-guide-with-vitis

[26] A. J. Landahl, J. T. Anderson, and P. R. Rice, "Fault-tolerant quantum computing with color codes," 2011, *arXiv:1108.5738*, doi: 10.48550/arXiv.1108.5738.

[27] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell, "Parallel window decoding enables scalable fault tolerant quantum computation," *Nat. Commun.*, vol. 14, no. 1, Nov. 2023, Art. no. 7040, doi: 10.1038/s41467-023-42482-1.

[28] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time," 2014, *arXiv:1307.1740*, doi: 10.48550/arXiv.1307.1740.

[29] R. W. J. Overwater, M. Babaie, and F. Sebastiano, "Neural-network decoders for quantum error correction using surface codes: A space exploration of the hardware cost-performance tradeoffs," *IEEE Trans. Quantum Eng.*, vol. 3, 2022, Art. no. 3101719, doi: 10.1109/TQE.2022.3174017.

[30] F. Battistel et al., "Real-time decoding for fault-tolerant quantum computing: Progress, challenges and outlook," *Nano Futures*, vol. 7, no. 3, Aug. 2023, Art. no. 032003, doi: 10.1088/2399-1984/aceba6.

[31] B. M. Terhal, "Quantum error correction for quantum memories," *Rev. Modern Phys.*, vol. 87, no. 2, pp. 307–346, Apr. 2015, doi: 10.1103/RevModPhys.87.307.

[32] D. Gottesman, "An introduction to quantum error correction and fault-tolerant quantum computation," 2009, *arXiv:0904.2557*, doi: 10.48550/arXiv.0904.2557.

[33] H. Bombín, "Topological codes," in *Quantum Error Correction*, D. A. Lidar and T. A. Brun, Eds., Cambridge, U.K.: Cambridge Univ. Press, 2013, pp. 455–481.

[34] X. Tan, F. Zhang, R. Chao, Y. Shi, and J. Chen, "Scalable surface-code decoders with parallelization in time," *PRX Quantum*, vol. 4, Dec. 2023, Art. no. 040344, doi: 10.1103/PRXQuantum.4.040344.

[35] T. Chan and S. C. Benjamin, "Actis: A strictly local union–find decoder," *Quantum*, vol. 7, Nov. 2023, Art. no. 1183, doi: 10.22331/q-2023-11-14-1183.

[36] M. J. Heer, E. D. Sozzo, K. Fujii, and K. Sano, "Novel union-find-based decoders for scalable quantum error correction on systolic arrays," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2023, pp. 524–533, doi: 10.1109/IPDPSW59300.2023.00092.