SpecMatching: Towards Real-Time Decoding of TeraQuop Regime Surface Codes

Sumeet Shirgure
University of Central Florida
Orlando, FL, USA
sumeetparameshwar.shirgure@ucf.edu

Yanzhang Zhu
University of Central Florida
Orlando, FL, USA
yanzhang.zhu@ucf.edu

Di Wu
University of Central Florida
Orlando, FL, USA
Di.Wu@ucf.edu

Siyuan Niu
University of Central Florida
Orlando, FL, USA
siyuan.niu@ucf.edu

arXiv:submit/8377756 [quant-ph] 15 Nov 2026

Abstract

Decoding the syndrome data generated during quantum error correction is essential to fault-tolerant quantum computing (FTQC). The leading candidate for fast and large-scale FTQC relies on superconducting qubits in a surface code, which produce syndromes every microsecond. Decoding must therefore run in real time with high throughput and low latency. The Minimum Weight Perfect Matching (MWPM) decoder is widely used for the surface code because of its accuracy. However, existing MWPM implementations do not meet the real-time requirements at the large distances necessary for billions of logical operations. We present **SpecMatching**, a speculative parallel MWPM design that accelerates decoding while maintaining exactness of matching. It decodes on a highly sparse speculated graph, which is rapidly built by a dedicated streaming preprocessor. It then solves MWPM on this graph in parallel, via divide-and-conquer and load balancing techniques.

Our architecture targets per round latency of less than a microsecond for distances up to $d = 31$ using a configuration of just 9 standard CPU cores and specialized streaming hardware for preprocessing, which significantly reduces the compute requirements compared with prior implementations. These estimates are obtained using simulated component latencies. We also demonstrate how our speculative framework can extend to decoding during lattice surgery while preserving exactness. With respect to the PyMatching (v2) baseline, we report a potential latency improvement of 2.0× at distance $d = 25$, and 2.5× at $d = 31$, under a circuit-level noise rate of $p = 10^{-3}$.

1 Introduction

Quantum computers promise to solve problems that are believed to be intractable for classical computers, such as factoring large integers [54] and simulating quantum many-body systems [22]. However, even the best superconducting processors today have error rates of order 10^{-3} per operation [28], while useful algorithms require billions of operations or more with a small total failure probability [5, 25]. Quantum error correction (QEC) bridges this gap by encoding logical qubits redundantly into many physical qubits [53, 60], and the threshold theorem guarantees that, if the physical error rate is below a threshold p_{th}, the logical error rate p_L can be made arbitrarily small by increasing the size of the code [1, 37, 39]. The surface code [16, 21] is the leading candidate for fault-tolerant quantum computing (FTQC) on superconducting hardware. It only requires nearest-neighbor interactions on a 2D grid, and it has a high threshold of about 0.6–1% under circuit-level noise [21, 32]. Logical operations between surface-code patches can be performed with lattice surgery [35]. Its logical error rate follows $p_L \propto (p/p_{\text{th}})^{(d+1)/2}$ in the code distance d [21], and recent experiments have demonstrated this error suppression [27, 28].

QEC also poses a classical problem. Every round of syndrome extraction takes about 1 μs on superconducting hardware [28]. The quantum processor measures a set of syndromes, and a classical *decoder* must infer the most likely errors from these measurements. If it processes syndromes at a lower throughput than with which they are produced, the backlog of un-decoded syndromes grows without bound and the computation stalls [4, 60]. Moreover, non-Clifford operations must wait for the decoder before they can proceed, so the reaction time of decoding sets the logical clock rate of the whole quantum computer [4]. In practice, decoders have been proposed on CPUs, GPUs [31, 33, 67], and on dedicated FPGAs and ASICs [3, 44, 66]. A large quantum computer with many logical qubits will require a significant number of decoders running in parallel, and each one adds power, area, and cost. Therefore, a decoder must achieve low logical error rates, meet real-time latency, and cost reasonable classical resources.

For the surface code, the most widely used decoder is minimum-weight perfect matching (MWPM) because of its accuracy [16] and readily available implementations [33, 67]. Other widely studied approximate decoders such as belief propagation [43, 48] and "pre-decoding" heuristics [2, 9, 42, 56, 64] can offer good empirical throughput. But their performance is often supported primarily by numerical evidence. In general, they do not provide the same fault distance preserving guarantees as exact matching. Even though these heuristics appear effective at small code distances or large noise rates, they may introduce rare failure mechanisms, unexpected phase transitions [31], or error floors [28], that prevent error suppression at larger distances or smaller noise rates [69].

A closely related alternative to MWPM is union-find based decoding [3, 14, 15, 44]. It offers near-linear computational complexity and is fast in practice. However, its existing rigorous threshold bounds are unphysically low [68], and it is measurably less accurate than MWPM [30, 65]. Its thresholds reach 0.6%, compared to 0.7% for MWPM [36]. Near the operating point $p \approx 10^{-3}$ [24, 28] the distance needed for a target logical error rate scales as $1/\log(p_{\text{th}}/p)$ [21], which means union-find will require about 8% larger distance,

or about 17% more physical qubits, or 27% more space-time volume than MWPM for the same error rate.

Exact MWPM retains the important advantage of being more accurate, and of having a well understood distance preservation and asymptotic fault-tolerance behavior [16, 19]. However, existing implementations trade latency against hardware cost, and no prior work achieves both at large distances. *Sparse Blossom* [33] reaches 1 μs per round on a single CPU core only up to $d \sim 20$ and is not parallelizable. *Fusion Blossom* [67] uses fifty to hundreds of CPU cores to achieve the needed throughput for similar distances. *Micro Blossom* [66] achieves the same on an FPGA-CPU co-design, but only up to $d = 13$, and requires $O(d^3)$ FPGA units for decoding of a single patch. Consequently, none of these approaches scales easily to the distances required in the gigaquop or teraquop regimes, which correspond roughly to $d = 25, 31$ at $p = 10^{-3}$ [47]. Moreover, evaluations in prior works are limited to memory experiments on a single logical qubit. They don't consider logical operations across multiple patches, such as with lattice surgery, which produce larger and more complex decoding problems [6, 55].

To address these challenges, we propose **SpecMatching**: a speculative parallel decoding framework that solves the MWPM problem exactly, whose design follows from the three key ideas given in Figure 1. It **speculates** on a sparse graph that likely contains the minimum-weight matching. For fast decoding, the speculated graph is built by a **dedicated streaming hardware**. If speculation fails, a fallback on the full detector graph runs in parallel, so the result is always exact. To parallelize decoding, we employ **divide-and-conquer** techniques and **load balancing**. Our models show that it can reach sub-microsecond average latency per round of syndrome extraction up to $d = 31$ at $p = 10^{-3}$ with just nine high performance CPU cores and a specialized streaming preprocessor. We also extend the design from single-patch memory circuits to logical operations using **lattice surgery**. In summary, we make the following contributions.

- We introduce a modified blossom algorithm that runs on a speculated graph $\mathcal{H}$, which has fewer edges than the original detector graph $\mathcal{G}$ at low error rates and is typically disconnected. If speculation fails, a "fallback" running in parallel on $\mathcal{G}$ ensures that MWPM is solved exactly.
- We design a graph-stream preprocessor that accelerates the construction of edges of $\mathcal{H}$.
- We propose a divide-and-conquer parallelization of blossom that balances the uneven sizes of components in $\mathcal{H}$ to improve latency.
- We extend our speculative framework to decoding across multiple surface code blocks during lattice surgery.

Our code is publicly available on GitHub [52].

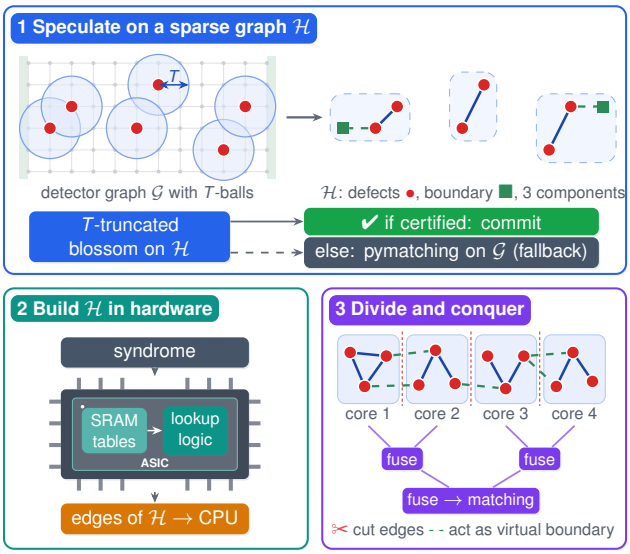


Figure 1: The three key ideas of SpecMatching. (1) A ball of radius T is placed around every defect of the detector graph $\mathcal{G}$ (left); we construct a sparse graph $\mathcal{H}$ over the defects, connected with edges only when the balls meet (right). $\mathcal{H}$ decomposes into independent connected components. If sparse blossom on $\mathcal{H}$ is successful, its matching is committed; otherwise *pymatching* on $\mathcal{G}$ is used. (2) Constructing $\mathcal{H}$ is reduced to lookups into tables and is offloaded to a streaming hardware preprocessor. (3) A large component of $\mathcal{H}$ is cut into pieces solved on separate cores, and the pieces are fused back along a fusion tree; lattice-surgery is handled similarly.

2 Background

2.1 QEC with Surface Codes

Quantum error correction [7, 50] with surface codes [20] uses a 2D lattice of physical qubits to encode one logical qubit. We assume basic familiarity with QEC, stabilizer codes [29], and detector error models (DEMs) [17, 23]. The DEM of a surface code is a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$: each vertex is a detector, and each edge is an error mechanism. An edge with error probability p_e has weight $w_e = \ln((1-p_e)/p_e)$. Each edge also carries one bit per observable which is 1 if the edge flips that observable [33]. The detectors that fire in a shot are the *defects* $\mathcal{D} \subseteq \mathcal{V}$. Sections 2.2 and 2.3 review the primal-dual MWPM algorithms we build on [11, 33, 38, 67].

2.2 Matching LP: Sparse Blossom

MWPM pairs up the defects, or matches them to a boundary vertex, along paths of minimum total weight. Sparse-blossom [33] (*pymatching*), is a software decoder that we use as a high performance single-thread MWPM solver. To make it compatible with our speculative framework, we make some modifications to sparse blossom that are detailed in Section 3.1.

We now describe the linear program (LP) formulation of matching. Let O be the family of vertex sets $S \in 2^{\mathcal{D}}$ that contain an odd number of defects (excluding the boundary), and let $\delta(S)$ be the set of edges with exactly one endpoint in S. The primal variable $x_e \in \{0, 1\}$ says whether edge e is selected. MWPM on detector graph $\mathcal{G}$ is the LP

$$\min_x \sum_{e \in \mathcal{E}} w_e x_e \qquad \text{subject to} \qquad \sum_{e \in \delta(S)} x_e \geq 1, \forall S \in O \qquad x_e \geq 0 \quad \forall e \in \mathcal{E} \tag{1}$$

whose dual is

$$\max_{y} \sum_{S \in O} y_S \qquad \text{subject to} \qquad \sum_{S \in O:\, e \in \delta(S)} y_S \le w_e, \forall e \in \mathcal{E} \qquad y_S \ge 0 \quad \forall S \in O \tag{2}$$

The primal constraint says that every odd set must be left by at least one matched edge. The boundary may match to any number of defects. Edmonds showed that this relaxation has an integral optimum [11, 38], so $x_e \ge 0$ may replace $x_e \in \{0, 1\}$.

A feasible (constraint satisfying) dual objective $\sum_S y_S$ is always a lower bound on the optimal weight of the matching (weak duality [11]), and hence acts as a *certificate of optimality* when the primal and the dual vectors have the same objective value. The dual also has a geometric meaning. A singleton $S = \{v\}$ with $y_v > 0$ is a *region*, a ball of radius y_v around defect v. A larger set S with $y_S > 0$ is a *blossom*, a shell of thickness y_S around the regions it contains. The dual constraint forbids regions and shells from overlapping on an edge. An edge is *tight* when the regions cover it completely. The blossom algorithm keeps a feasible dual and a partial matching on tight edges, and it grows and shrinks the y_S until the matching is perfect. The matched and growing regions are organized in *alternating trees*. For a defect u, the quantity of interest is the *nested dual sum*

$$Y(u) = \sum_{S \in O:\, u \in S} y_S, \tag{3}$$

the radius, measured from u, that the outermost region containing u has reached (the balls of Figure 1). Let $d_{\mathcal{G}}(u, v)$ be the shortest-path distance in $\mathcal{G}$, and $b_{\mathcal{G}}(u)$ the distance from u to the nearest boundary. Two matched defects satisfy $Y(u) + Y(v) \ge d_{\mathcal{G}}(u, v)$, and a defect matched to the boundary satisfies $Y(u) \ge b_{\mathcal{G}}(u)$.

2.3 Divide and Conquer: Fusion Blossom

Fusion Blossom [67] runs the same primal-dual algorithm, but stores its state differently. Sparse Blossom stores each radius implicitly, as a function of one global clock variable, and schedules all events on one priority queue. Fusion Blossom stores every dual variable explicitly, so its state is just a feasible primal-dual pair (x, y), and it can resume from *any* such pair. This makes it parallelizable. It divides $\mathcal{G}$ into sub-graphs along a set of vertices $\mathcal{V}_b$, and solves each sub-graph independently, with $\mathcal{V}_b$ acting as a boundary that absorbs any number of defects. A *fusion* then joins two solved sub-graphs. It removes their matches to the cut, turns the released defects into new alternating trees, and resumes the solver, keeping all other matches and duals. A defect matched to a nearby cut has explored little, so a fusion touches only the defects near the cut. The sub-problems and the binary tree of fusions over them can be scheduled on parallel workers (Figure 1(3)).

3 Motivation and Insights

This section motivates the three ideas of Figure 1 by providing the numerical evidence shown in Figure 2.

3.1 Speculative Matching On a Sparse Graph

pymatching is serial, and its runtime grows with the space-time volume of $\mathcal{G}$. Figure 2(a) shows that, as d grows, it eventually takes longer than the 1 μs syndrome cycle per round. Yet the defects are sparse, and they mostly form small, well-separated clusters (Figure 1(1)). Our first idea leverages the fact that sparse blossom rarely explores the regions outside these clusters.

Table 1: Fallback rates $q(d, p, T)$ of the sparse-graph pre-filter on rotated surface-code memory, by code distance d, physical error rate p, and cutoff T. (Estimates over a million shots.)

p	d	$T = 1$	$T = 3/2$	$T = 2$	$T = 5/2$
5×10^{-4}	17	2.81×10^{-2}	3.46×10^{-3}	2.20×10^{-5}	4.00×10^{-6}
	25	7.26×10^{-2}	7.69×10^{-3}	7.70×10^{-5}	8.00×10^{-6}
	31	1.25×10^{-1}	1.21×10^{-2}	1.39×10^{-4}	1.40×10^{-5}
10^{-3}	17	9.63×10^{-2}	1.34×10^{-2}	2.54×10^{-4}	2.50×10^{-5}
	25	2.48×10^{-1}	3.10×10^{-2}	5.37×10^{-4}	5.80×10^{-5}
	31	4.04×10^{-1}	4.90×10^{-2}	9.09×10^{-4}	1.11×10^{-4}
3×10^{-3}	17	5.83×10^{-1}	7.84×10^{-2}	6.08×10^{-3}	6.65×10^{-4}
	25	9.28×10^{-1}	1.85×10^{-1}	1.58×10^{-2}	1.87×10^{-3}
	31	9.92×10^{-1}	2.88×10^{-1}	2.58×10^{-2}	3.04×10^{-3}

We thus build a sparse graph $\mathcal{H}(\mathcal{D}, \mathcal{E}_{\mathcal{H}})$ whose vertices are the defects alone. Its edges join nearby defects, with their distance $d_{\mathcal{G}}$ as weight. Figure 3 shows an example construction. $\mathcal{H}$ is much sparser than $\mathcal{G}$ at the noise rates we target ($p \approx 10^{-3}$). This makes matching on $\mathcal{H}$ faster because it doesn't have edges and detectors that can't be a part of the final matching. We call this matching *speculative*, because it assumes that the optimal matching pairs only nearby defects. A modified *pymatching* solves $\mathcal{H}$ and either proves that its matching is optimal on $\mathcal{G}$ (via the constructed dual y), or signals that it cannot prove it so. In the latter case the shot is decoded by plain *pymatching* on $\mathcal{G}$, which has run in parallel from the start, much like speculative execution in a processor. This exact fallback mechanism ensures optimality and accuracy of matching.

The structure of $\mathcal{H}$ is determined by the *dual cutoff* 'T'. For every defect, we cap the nested dual sum $Y(u) \le T$ (Equation 3), in units of the typical edge weight $w_p = \ln((1-p)/p)$. So T is roughly the number of edges the solver may explore from a defect. The cap applies to $Y(u)$ rather than to each y_S, so it also bounds the region of a defect inside a blossom. A *T-truncated blossom* is sparse/fusion blossom with this cap on $Y(u)\ \forall u \in \mathcal{D}$. When it cannot match every defect within the cap, it stops with a *fallback event*. An example syndrome that triggers fallback is just two isolated defects at the ends of an error chain longer than $2T$, with no other defects nearby. Under the cap, two defects can only be matched if $d_{\mathcal{G}}(u, v) \le 2T$, and a defect can only reach the boundary if $b_{\mathcal{G}}(u) \le T$. Thus, the graph with edge set

$$\begin{aligned} \mathcal{E}_{\mathcal{H}}(T) = \{(u, v) : u, v \in \mathcal{D},\ d_{\mathcal{G}}(u, v) \le 2T\} \\ \cup \{(u, \partial) : u \in \mathcal{D},\ b_{\mathcal{G}}(u) \le T\}, \end{aligned} \tag{4}$$

weighted by $d_{\mathcal{G}}$ and $b_{\mathcal{G}}$, holds all the information that a T-truncated blossom on $\mathcal{G}$ uses, and a T-truncated blossom on $\mathcal{H}$ behaves exactly like one on $\mathcal{G}$. Geometrically, $\mathcal{H}$ is built by placing a ball $B_T(u)$ of radius T around every defect, and links two defects when their balls meet, or when a defect and the boundary when its ball reaches it (Figure 3). The balls are precomputed and cached once per DEM in *ball tables* (Section 4.1.1).

Table 1 gives the fallback rate $q(d, p, T)$ for sample settings for a d-round memory experiment. The results show that q increases with d and p and decreases with T. Clearly $q \to 0$ when $T \to \infty$.

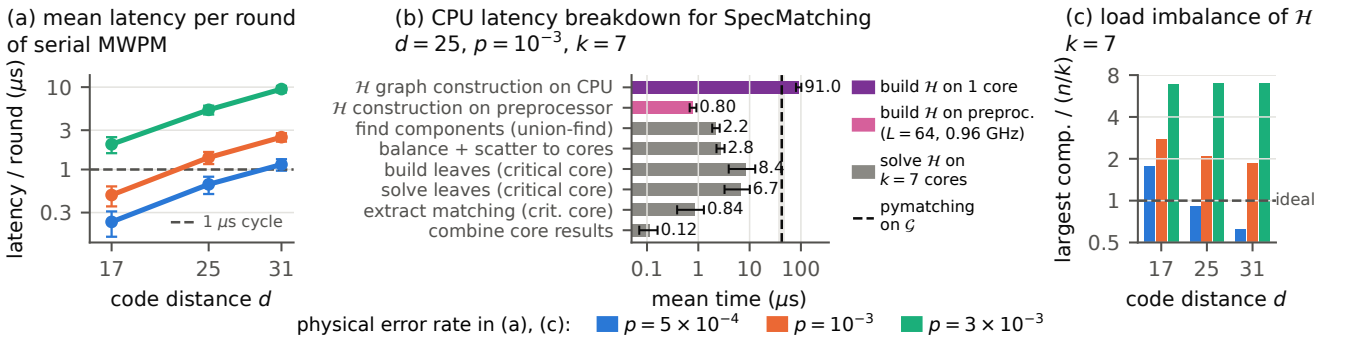


Figure 2: The three motivations behind SpecMatching. Every latency is the CPU time of a single thread of an Apple M5 Pro ($\leq$ 4.6 GHz). (a) Mean latency of *pymatching* on the full detector graph $\mathcal{G}$, divided by the number of rounds. (b) Mean time of each stage of SpecMatching without divide and conquer, (5×10^5 shots). The construction of $\mathcal{H}$ is performed on one CPU core and on the graph-stream preprocessor of Section 4.1. The dashed line is the mean latency of *pymatching* on $\mathcal{G}$. (c) Mean imbalance factor of $\mathcal{H}$ at $T = 2$, $k = 7$ solver cores.

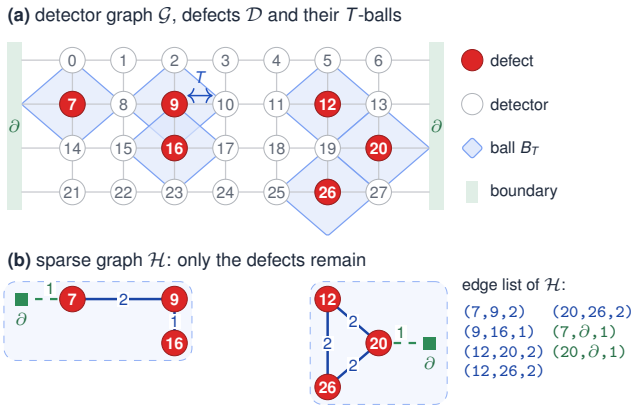


Figure 3: An example of the construction of $\mathcal{H}$ with $T = 1$ and unit edge weights. (a) The detectors are numbered 0–27, and ∂ is the boundary. Each of the six defects is shown with its ball B_T. (b) $\mathcal{H}$ connects two defects u, v if $d_{\mathcal{G}}(u, v) \leq 2T$ (their balls meet), and a defect u to ∂ if $b_{\mathcal{G}}(u) \leq T$ (its ball reaches the boundary). The edge list on the right is what the preprocessor of Section 4.1 emits. The minimum-weight matchings of the two components are $\{(7, \partial), (9, 16)\}$ and $\{(12, 26), (20, \partial)\}$.

Increasing p raises the chances of longer error chains. And a larger volume of syndromes ($\propto d^3$) raises the chances of a fallback event happening somewhere in it. Increasing T adds edges to $\mathcal{H}$ roughly proportionally to $\sim T^3$ and burdens the speculative solver. But decreasing T raises the fallback rate q, and with it the average latency of the composite system when fallbacks are slower. So we minimize T as much as possible while incurring acceptable fallback rates. We choose $T = 2$, giving $q < 10^{-3}$ for the (d, p) of interest.

3.2 Building $\mathcal{H}$ Is On the Critical Path

Solving MWPM on $\mathcal{H}$ is faster, but it costs us time to construct $\mathcal{H}$ in the first place. For every defect, we must find the other defects within distance $2T$ in $\mathcal{G}$. Even if we cache shortest paths offline, constructing $\mathcal{H}$ serially on CPU is memory bound and costs a lot of latency. Figure 2(b) shows the breakdown of the latency of SpecMatching stages when the components of $\mathcal{H}$ are solved on $k = 7$ cores. Building $\mathcal{H}$ on one CPU core takes 91.0 μs at $d = 25$, more than everything else combined, which removes any speedup. However, the edges for different pairs of defects are independent, so they can be built in parallel. Therefore, we design a streaming preprocessor (Section 4.1) which can build them in 0.80 μs for the same d, achieving a 114× speedup.

3.3 Percolation Imbalances the Load

$\mathcal{H}$ splits into connected components (Figure 3(b)), and a T-truncated blossom can solve each one independently. A natural parallelization therefore distributes the components over k solver cores. The runtime of blossom is roughly linear in the number of defects $n = |\mathcal{D}|$ [33, 67], so each core should receive about n/k defects. But we observed that the distribution of component sizes is imbalanced. As p and T increase, the balls of nearby defects start to overlap and *percolate*. This leads to many small connected components and a few disproportionately large ones. Figure 2(c) shows that, for $k = 7$, the largest component is on average twice the ideal load, and about seven times at $p = 3 \times 10^{-3}$. The core that receives the largest component typically finishes last, while the others sit idle.

Large components must therefore be split across solvers and recombined. Fusion Blossom (Section 2.3) is exactly the tool to do that: divide components, solve the parts independently, and fuse. We divide a large component of $\mathcal{H}$ along *edge cuts*, defined by a set of *crossing edges* whose removal splits it into *pieces* of bounded size (Figure 1(3)). Since the preprocessor delivers edges, we can compute edge cuts easily (detailed in Section 4.2.1). Each defect u on a crossing edge gets a *dummy boundary* edge of weight $b(u) = w_{\min}(u)/2$, where $w_{\min}(u)$ is the weight of its lightest crossing edge. This dummy plays the role of the cut in Fusion Blossom. Blossom obeys dual constraints, so $Y(u) \leq b(u)$, and for any crossing edge (u, v, w), $Y(u) + Y(v) \leq \frac{1}{2} w_{\min}(u) + \frac{1}{2} w_{\min}(v) \leq w$. The duals of

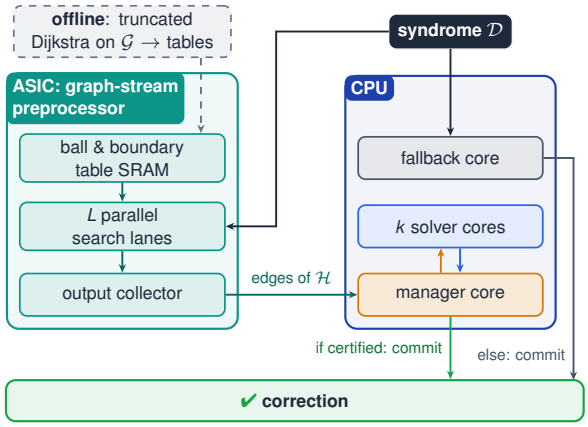


Figure 4: The two subsystems of SpecMatching. Offline, T-truncated Dijkstra on the detector graph $\mathcal{G}$ compiles the ball and boundary tables into the SRAMs of the preprocessor. Online, the syndrome $\mathcal{D}$ of each shot goes to the ASIC preprocessor and the fallback core. The preprocessor streams the edges of $\mathcal{H}$ to the manager core. If the speculation is successful, it is committed as soon as it is ready; otherwise the fallback's is. Figure 6 details the roles of the CPU cores.

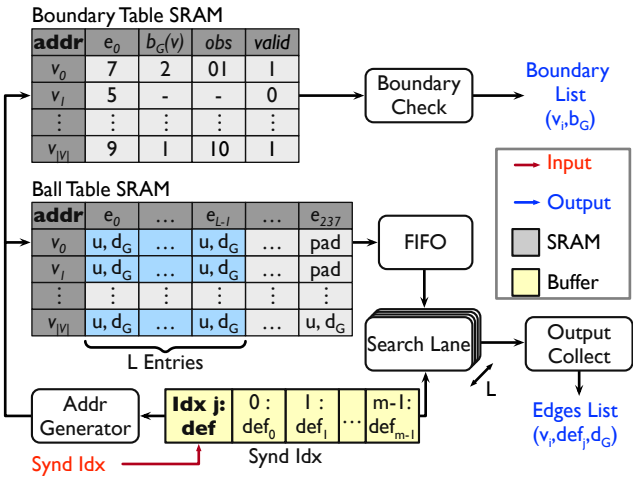


Figure 5: Hardware architecture of streaming preprocessor.

the two pieces are thus jointly feasible. Intuitively, the dummies restrict the growth of regions next to a cut, and fusion lifts the restriction later. Fusion then proceeds as in Section 2.3, with the cap T still applied to $Y(u)$. Divide and conquer returns the same matching/fallback signal as if a T-truncated blossom ran on the whole component sequentially.

4 Proposed SpecMatching Architecture

Section 3 reduces exact decoding to three tasks: build $\mathcal{H}$, solve MWPM on its pieces with T-truncated blossom, and solve the same on $\mathcal{G}$ in case a fallback is needed. SpecMatching gives each task its own component (Figure 4), with some tasks that can be done offline before decoding begins. Offline, two lookup tables are compiled from $\mathcal{G}$: the *ball table* and the *boundary table*. Online, each syndrome goes to the preprocessor and the fallback core. The ASIC preprocessor uses the tables to build $\mathcal{H}$ and feeds it to the manager core, which partitions it and dispatches jobs to k solver cores. Meanwhile, the fallback core decodes the same syndrome on $\mathcal{G}$. If the speculative part of the decoder succeeds, it is committed as immediately; otherwise, the decoder waits for the fallback core.

4.1 Streaming Preprocessor

Building $\mathcal{H}$ on a CPU core takes longer than solving it (Section 3.2), which puts it on the critical path. The preprocessor removes this bottleneck by splitting the work into an offline stage, which precomputes every edge that $\mathcal{H}$ can need, and an online stage, which only looks them up.

4.1.1 Ball table and Boundary table construction. The weights of $\mathcal{H}$ are shortest-path distances in $\mathcal{G}$ (see Figure 3 for examples). Since $\mathcal{G}$ is fixed for a given DEM, we compute them once, ahead of time. For every non-boundary vertex u, a Dijkstra search [18] records two things: in the ball table $Ball[u]$, every detector v within distance $2T$ of u, with shortest-path distance $d_{\mathcal{G}}(u,v)$; and in the boundary table $Bnd[u]$, the distance $b_{\mathcal{G}}(u)$ to the nearest boundary, if it is at most T. Dijkstra settles vertices in order of distance, so each search stops once it passes $2T$. This yields all the edges that Equation 4 can need. For a vertex u, we call the output of its search the *ball table* row of u: the pairs $(v, d_{\mathcal{G}}(u,v))$ over the vertices $v \neq u$ with $d_{\mathcal{G}}(u,v) \leq 2T$, and the boundary distance $b_{\mathcal{G}}(u)$ if $b_{\mathcal{G}}(u) \leq T$ which is stored in its boundary table row. The maximum size of each row of the ball and boundary tables depends on T and on the noise model, but not on the number of syndrome rounds.

A table with one ball per detector grows as $O(T^3 d^3)$. But most balls are copies of each other shifted in time, since the neighborhood of a stabilizer looks the same in every round. We therefore store each ball entry as a weight and an *offset* $\delta = v - u$ from its source. Two detectors whose lists of (δ, w) are identical share one row of the ball table. For example, in Figure 3, the $T = 1$ ball table rows for defects 9 and 16 can be the same. In which case, the same entry in the row stores the edges $(9, 10, w_{9,10})$, and $(16, 17, w_{16,17})$ as an offset $(\delta = 1, w_{9,10})$ The number of rows per position is set by the ball radius, so the tables grow only as $O(T^4 d^2)$. A DEM that has a lot of variation just yields more rows, but the result is unchanged.

4.1.2 Subgraph construction. Section 4.1.1 moves offline the expensive part of building $\mathcal{H}$, and leaves the streaming preprocessor a single online step: turning each syndrome into the edge list of $\mathcal{H}$ (Figure 5). It consists of the two table SRAMs, an address generator, a FIFO, L parallel search lanes ($L = 64$ in our design), an output collector and a boundary-check path. The address generator walks the sorted list of defects. For example, in Figure 3 the list is $\{7, 9, 12, 16, 20, 26\}$. For each defect def_i it reads the boundary table word, addressed by detector id, which holds the row pointer, the row length and the boundary entry of def_i. It then streams the row from the Ball Table SRAM, one chunk of L entries per cycle. The FIFO aligns each chunk with the id of its source and absorbs stalls. Each lane computes the target $u = def_i + \delta$, drops padding entries (stored as $\delta = -1$) by their sign bit, and binary-searches u in the

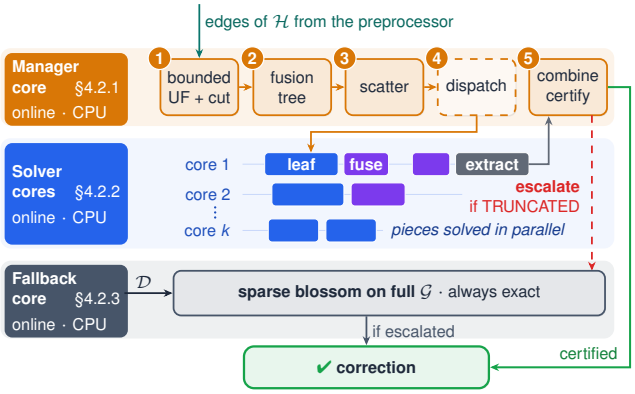


Figure 6: Roles of the CPU cores. The manager receives the edges of $\mathcal{H}$ from the preprocessor, cuts its oversized components into pieces of at most $\lceil n/k \rceil$ defects, builds the fusion tree, groups the edges by piece, dispatches leaf, fusion and extraction jobs to the k solver cores, and combines and certifies their results. If any job reaches the cutoff T, the shot escalates and the result of the fallback core is used.

sorted defect list, in $O(\log m)$ steps for m defects. A hit yields an edge (def_i, def_j) of weight $d_{\mathcal{G}}(def_i, u)$. All L lanes may hit in the same cycle, so the collector buffers the hits of a chunk and emits one edge per cycle. In parallel, the boundary-check path emits the boundary edge of def_i on its own port. This organization reduces runtime preprocessing to SRAM reads, parallel binary searches, and output serialization. Binary search avoids the large comparator array of a fully associative match. These choices favor a simple datapath with predictable latency.

4.2 CPU Cores

Once the edges of $\mathcal{H}$ arrive from the preprocessor, what remains is to solve MWPM on $\mathcal{H}$ in parallel and extract the result. This section explains the responsibilities of the individual cores in Figure 6, namely the manager, solver and fallback cores.

4.2.1 Manager core. The components of $\mathcal{H}$ can be very uneven in size (Section 3.3), so assigning whole components to cores is slow. The manager therefore cuts large components into pieces of bounded size and schedules their solves and fusions. It receives the edge list from the streaming preprocessor, and it divides $\mathcal{H}$ into pieces that the solver cores can solve independently. The manager also orchestrates the fusion tree. Let $n = |\mathcal{D}|$ be the number of defects, and let $n_{\max} = \lceil n/k \rceil$ be the target load of a solver core. The manager's duties are the following.

Step 1: Bounded union-find. The manager runs a disjoint set union-find over the edges of $\mathcal{H}$ to find components. For an edge (u, v) between different sets $A \ni u$ and $B \ni v$, it merges the sets only if $|A| + |B| \le n_{\max}$; otherwise (u, v) becomes a *crossing edge*. At the end, each set is a *piece* of at most $n_{\max}$ defects. A small component is never cut, and a large one is split into pieces joined by crossing edges. This single pass thus finds both the components and the cuts. Each endpoint u of a crossing edge gets the dummy boundary of Section 3.3, $b(u) = \min(w_b(u), \lfloor w_{\min}(u)/2 \rfloor)$, where $w_b(u)$ is the weight of its real boundary edge (∞ if none) and $w_{\min}(u)$ is a running minimum over its crossing edges.

Step 2: Fusion tree. The pieces of each large component are reassembled along a binary *fusion tree*, as in [67] (Figure 1(3)). The manager builds it greedily, repeatedly merging the two adjacent nodes with the fewest crossing edges between them. Each crossing edge belongs to the tree node where both of its endpoints first meet.

Step 3: Scatter. The manager reorders the edge list by grouping the edges in a piece in contiguous subsegments. Each solver core can therefore build its piece's adjacency list in parallel, without scanning the whole edge list. Steps 1–3 take $O((n+m)\,\alpha(n))$ time for m edges, where $\alpha(n)$ is the inverse Ackermann function [59].

Step 4: Dispatch. The pieces and the fusion tree define a graph of jobs that are sent to the solver cores (Section 4.2.2). Every piece is a *leaf* job, every internal node of the fusion tree is a *fuse* job, and every root an *extract* job (blue, purple and grey solver core jobs in Figure 6). A job is ready when its children have finished. The manager hands the most expensive ready job to the next idle core (list scheduling). A fusion runs on the core of the child that finished last, which already holds half of its state necessary to resume a T-truncated sparse blossom.

Step 5: Combine and certify. The manager combines the observables and weights of all extract jobs' results received from the solver cores into the overall correction. If any solver job raises the TRUNCATED flag (Refer to Section 4.2.2 for the definitions of extract jobs and the TRUNCATED flag) the shot *escalates* to the fallback core (Section 4.2.3).

4.2.2 Solver cores. The solver cores run T-truncated blossom on the pieces of $\mathcal{H}$. They also resume partially completed solutions on fusions across pieces. It is sparse blossom with two additions. First, an edge can be *masked*, which hides it from the solver. Second, a job that would let some $Y(u)$ exceed T stops and reports TRUNCATED. All other operations are those of sparse/fusion blossom. The solver state of a shot lives in one shared memory area indexed by defect. The jobs can either be *leaf* jobs, *fusion* jobs or *extract* jobs.

A *leaf job* builds the adjacency list of its piece, including the masked crossing edges and the dummy boundaries, and solves it. With the crossing edges masked, this is an exact MWPM on the piece, in which the cut acts as a boundary. A *fuse job* first unmasks the crossing edges between its two sub-problems. For every endpoint u with a dummy boundary, it recomputes $b(u)$ over the crossing edges that still leave the fused subtree; $b(u)$ can only grow, and the dummy node (see Section 3.3) disappears when no crossing edge remains. If u was matched to its dummy, that match is removed and u becomes the root of a new alternating tree, keeping its radius. The solver then resumes from the combined state, keeping all other matches and duals. An *extract job* reads off the matched pairs of a root of the fusion tree and returns their weight and observables to the manager core (Section 4.2.1).

4.2.3 Fallback core. Speculation can fail when some defect needs a region larger than the cutoff T allows (Table 1). The parallel fallback core keeps the decoder exact in that case, without delaying the other shots. If all jobs complete, the manager commits the matching at once. If any job reports TRUNCATED, the decoder discards the speculative result and commits the result of the fallback core, which runs unmodified *pymatching* on $\mathcal{G}$ from the start of every shot. The

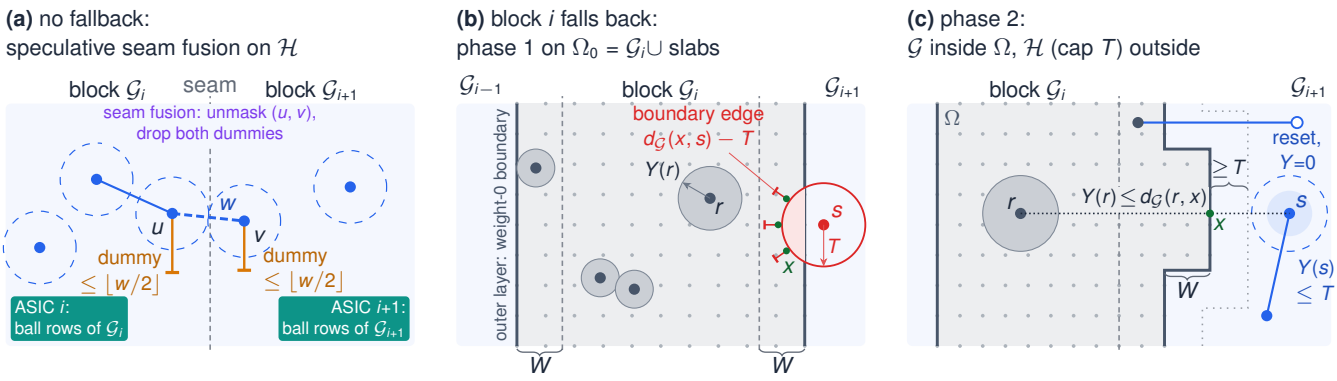


Figure 7: Decoding a lattice-surgery experiment with one SpecMatching instance per block. The dot grid is the full-resolution detector graph $\mathcal{G}$; elsewhere the decoder sees only the defects and the edges of $\mathcal{H}$. (a) If no block falls back, fusion over $\mathcal{H}$ combines blocks across seams. The preprocessor (ASIC) of each block stores only the ball-table rows of its own detectors. (b) If block i falls back. Phase 1 solves the defects of Ω_0, the block plus a slab of width W of each neighbor, with uncapped blossom on $\mathcal{G}$. A neighboring defect s outside Ω_0 whose ball of radius T reaches into Ω_0 is an obstacle: every shell node x of its ball gets a boundary edge of weight $d_{\mathcal{G}}(x, s) - T$, so the region of every defect r stays clear of the largest speculative region s can have. (c) Phase 2 resumes on $\mathcal{G}$ inside the territory Ω and on $\mathcal{H}$, with the cap T, outside it. A speculative component that reaches into Ω, or comes within T of it, is reset to zero duals and absorbed into Ω.

latency of a shot is therefore

$$t_{\text{sys}} = \begin{cases} t_{\text{fb}} & \text{if the shot escalates,} \\ \min\left(t_{\text{fb}},\ t_{\text{spec}}\right) & \text{otherwise,} \end{cases} \tag{5}$$

where t_{fb} is the latency of the fallback and t_{spec} that of the speculative path, from the arrival of the syndrome to the end of Step 5. Since the escalation rate q is typically small, the mean latency is close to that of the speculative path, while no shot is ever slower than *pymatching* alone. Speculation affects only the latency, never the correctness of the decoder.

5 Lattice Surgery

So far we have decoded a memory experiment, which keeps one logical qubit alive for d rounds of syndrome extraction. Computation also needs logical qubits to interact, most commonly through lattice surgery [57]. In this section we discuss the application of SpecMatching instances to multiple surface code patches that undergo a lattice surgery operation.

SpecMatching's design is *composable*. A large computation can be decoded by tiling SpecMatching instances, one per block of $\Theta(d^3)$ syndromes. The speculative part of the instances communicate only at the *seams* between blocks. We define seams to be boundaries that our tiling introduces, and that are not boundaries of the DEM. For our evaluation, we consider a lattice-surgery "merge" operation on N patches in a strip, each with distance-d, where adjacent patches are jointly measured for d rounds. We cut $\mathcal{G}$ into blocks $\mathcal{G}_1, \ldots, \mathcal{G}_N$, one per patch, and decode each with its own SpecMatching instance (preprocessor, manager, k solver cores and a fallback core). An extra *global manager* core coordinates the N SpecMatching instances. Time-like seams, between consecutive windows of syndrome rounds, can be handled the same way.

Each instance should store and build only its own part of $\mathcal{H}$, so that its hardware does not grow with N. The preprocessor of block i therefore stores only the ball-table rows of the detectors of $\mathcal{G}_i$. The ball of a defect near a seam reaches into the neighboring block, so the preprocessor also emits the *cross-seam* edges between its defects and those of the adjacent blocks (Figure 7(a)). Each SpecMatching manager treats them exactly as crossing edges (Section 4.2.1), so joining two blocks is an ordinary fuse job. Unlike parallel window or modular decoding [6, 55], no buffer region is decoded twice, and the fused matching is exact.

The above discussion applies in the case when there is no fallback event, which is the case for most shots. A fallback is a property of a single component of $\mathcal{H}$ when it flags TRUNCATED. So when a component inside block $\mathcal{G}_i$ truncates, the other components of the strip still complete and fuse on $\mathcal{H}$. Escalating the whole strip to *pymatching* on $\mathcal{G}$ would give up the speedup of every block for a rare event in one of them. This motivates us to keep the fallback local. The block $\mathcal{G}_i$ is decoded from its fallback core, in two phases. Let r denote a defect solved on $\mathcal{G}$ by that fallback core, s a defect of a neighbouring block, $Y(\cdot)$ the nested dual sum of Equation 3, and $d_{\mathcal{G}}$ the distance in the whole $\mathcal{G}$.

Phase 1 runs from the start, like the fallback of Section 4.2.3. Let Ω_0 denote the initial *territory*, which spans the block $\mathcal{G}_i$ together with the detectors within W of $\mathcal{G}_i$ (Figure 7(b)). The fallback core now solves every defect r in Ω_0, including those in two W-wide slabs reaching into adjacent blocks. The fallback runs uncapped blossom on $\mathcal{G}$. The fallback core cannot know the speculative state of the defects in the neighboring region, but only that every such defect s has $Y(s) \leq T$. It therefore treats each defect s outside Ω_0 whose ball $B_T(s)$ reaches into Ω_0 as an *obstacle*, which is a frozen ball of radius T. It adds constraints (via dummy boundary edges in Figure 7(b)) so that sparse blossom on Ω_0 stays clear of obstacles. Thus, the state of the fallback is compatible with *any* speculative state of its neighbors.

Phase 2 joins the fallback state with the speculative state of the rest of the graph (Figure 7(c)). Intuitively, the territory Ω is the region where the full resolution of $\mathcal{G}$ is needed to finish the exact solve. Inside Ω, the decoder works on $\mathcal{G}$, with explicit detectors and uncapped regions, just like the fallback core. Outside it, everything stays at the resolution of $\mathcal{H}$. Ω starts as Ω_0, and it only grows, where the solve pushes against its boundary. As Ω grows, external defects touching it are reset by setting their dual to zero, and are later "absorbed" into Ω. In the worst case Ω grows to all of $\mathcal{G}$ and the solve is plain blossom. But as for any decoder, the locality of its solve is highly likely, and a fallback is typically resolved without escalating the whole computation from $\mathcal{H}$ to $\mathcal{G}$.

6 Evaluation

We evaluate SpecMatching on the following questions.

- **Decoding latency** (Section 6.2): How much faster than *pymatching* is SpecMatching? Can it keep up with a 1 μs syndrome cycle in a teraquop regime? When is divide and conquer needed? What is the latency breakdown of the critical path?
- **Comparison with other exact matching decoders** (Section 6.3): How does SpecMatching compare with Fusion Blossom [67] and Micro Blossom [66]?
- **Lattice surgery** (Section 6.4): How does SpecMatching compare to *pymatching* on a lattice surgery DEM ?
- **Area and power** (Section 6.5): What does the preprocessor cost in area and power, compared with Micro Blossom?

6.1 Experimental Setup

6.1.1 Decoding latency. We use rotated surface-code memory experiments with d rounds of syndrome extraction, generated by *Stim*, under a uniform circuit-level noise of strength p. We use $d \in \{17, 25, 31\}$ and $p \in \{5 \times 10^{-4}, 10^{-3}, 3 \times 10^{-3}\}$. The dual cutoff is $T = 2$ in w_p units, and the number of solver cores is $k \in \{4, 7, 10\}$. For every (d, p) we decode 5×10^5 shots. Each experiment is run after 10^3 warm-up shots to bring the cache and the branch predictors to a steady state. The fallback core runs stock *pymatching* on $\mathcal{G}$, and the fallback's performance is also a baseline.

We **do not** run SpecMatching on real parallel threads. Instead, every step of Section 4, except the construction of $\mathcal{H}$, is executed and timed on a single thread of an Apple M5 Pro ($\leq$ 4.6 GHz). We then place the measured durations on a simulated list schedule with one manager core and k solver cores. The makespan of this schedule is t_{spec}. Together with the latency of the fallback, t_{fb}, it gives the system latency t_{sys} of Equation 5. The latency of the preprocessor, $t_{\mathcal{H}}$, comes from its cycle model (described below) and is added on top. Since $\min(t_{\text{fb}}, t_{\mathcal{H}} + t_{\text{spec}}) \leq \min(t_{\text{fb}}, t_{\text{spec}}) + t_{\mathcal{H}}$ on every shot, we report the upper bound $\min(\bar{t}_{\text{fb}}, \bar{t}^{\text{CPU}}_{\text{sys}} + \bar{t}_{\mathcal{H}})$ on the mean where bars denote averages of shots.

The first experiment profiles this mean latency on every (d, p) above and compares it with *pymatching* on one core. We report the mean because of two reasons. First, in a memory experiment, the syndrome rounds that are not yet decoded fill a queue. This queue is stable if and only if the *mean* decoding time is below the time between shots [45, 60]. Second, the resource estimates of parallel window decoding [55, 58] also depend only on the mean decoding time per window. The tails also matter, because they set the reaction time of the decoder. So we also show the per-shot distributions.

To measure how much divide and conquer contributes, we also time the t_{spec} without it. In this ablation, the manager finds the components of $\mathcal{H}$ with an unbounded union-find and assigns whole components to the k solver cores, largest first (Section 3.3). The solver cores then solve these components without any cut or fusion. The ablation runs on a different set of 5×10^5 shots per (d, p), for $p \in \{5 \times 10^{-4}, 10^{-3}\}$, and is also simulated on one core.

The preprocessor latency $t_{\mathcal{H}}$ comes from a cycle model of its RTL. We cross-check this model against Verilator simulation and against the software construction of $\mathcal{H}$, and run it with $L \in \{16, 32, 64, 128\}$ search lanes over 10^4 shots per (d, p). The latencies reported below use the time-shift tables of Section 4.1.1 with $L = 64$ lanes. We clock the preprocessor at the post-route maximum frequency of its critical block, the search lane: 1.08 GHz at $d = 17$, 0.96 GHz at $d = 25$, and 0.86 GHz at $d = 31$. The last value is the pessimistic end of the 0.86–0.91 GHz range that we extrapolate from smaller distances. For the per-shot distributions of Figure 8, we instead use a linear model of the cycle count in the numbers of defects and edges of $\mathcal{H}$. We fit this model to the nine (d, p) means, and it reproduces them within 2.4%.

6.1.2 Comparison with other decoders. The second experiment compares SpecMatching with two exact parallel MWPM decoders, Fusion Blossom and Micro Blossom. We run Fusion Blossom [67] on 10^5 shots of the same $\mathcal{G}$. We cut $\mathcal{G}$ along time into $P \in \{1, 2, 4, 8\}$ blocks (also 16 at $d = 31$) and fuse the blocks along a balanced tree. As for SpecMatching, we time every leaf and every fusion on one thread and place them on the same list scheduler. Fusion Blossom gets all nine cores of SpecMatching as solver cores. For every (d, p) we report the best P, without the time Fusion Blossom spends reconstructing the correction (24 μs to 1.4 ms).

For Micro Blossom [66], we use its final configuration on the same $\mathcal{G}$: batch decoding with pre-matching and layer fusion, and 4-bit weights $2\lceil 7w/w_{\max} \rceil$. We don't simulate its RTL at $d = 31$ because it has 30k vertices and 169k edges. We therefore estimate its latency on 5×10^3 shots per (d, p) at $p \leq 10^{-3}$. The estimate adds three parts, (i) The CPU part is measured: we run its unmodified primal code on one core of the same M5 Pro. (ii) The responses of the accelerator come from a behavioral model of the array, which produces instruction traces byte-identical to Micro Blossom's simulator. (iii) The accelerator time is modeled as 10 cycles at 1 GHz per accelerator instruction, plus a fixed round-trip time (RTT) for every exchange between the CPU and the accelerator.

6.1.3 Lattice surgery. The third experiment decodes a $1 \times N$ strip of distance-d patches, $N \in \{2, 4, 8, 16\}$, merged by lattice surgery for d rounds (Section 5). We generate the circuits with a custom Stim generator that uses the same noise channels as the memory experiments. We use $d \in \{17, 25\}$, $p \in \{5 \times 10^{-4}, 10^{-3}\}$, $k \in \{4, 7\}$ and slab width $W = 2$, with 10^5 shots per configuration. As in Section 5, every patch has its own SpecMatching instance of $k + 2$ cores, and a global manager coordinates them. Every patch also has its own preprocessor, so we approximate their latency by the mean preprocessor latency of a memory experiment with the same (d, p). We check every shot against *pymatching* on the whole strip.

6.1.4 Area and power. The last evaluation estimates the silicon cost of the preprocessor. We synthesize and place-and-route it in the ASAP7 7 nm predictive PDK [10] for 1 GHz at 0.77 V. Logic area and power are taken from Genus and from the post-route reports, at the tool's default (vectorless) switching activity. All SRAMs are modeled with FN-CACTI's 7 nm FinFET device at 0.7 V[1]. The $d = 31$ point is a projection, not a direct measurement. For the logic, we fit power laws to place-and-route runs at smaller distances and extrapolate them, while FN-CACTI models the SRAM at the exact $d = 31$ geometry. For comparison, we also synthesize and place-and-route Micro Blossom [66] from its open-source RTL at the same d, with 16-bit edge weights instead of 4.

6.2 Decoding Latency

PyMatching baseline. At $p = 10^{-3}$ and $k = 7$, with the preprocessor included, it is 2.04× faster at $d = 25$ and 2.51× faster at $d = 31$. The speedup thus *grows* with d. A memory experiment at distance d has d rounds, so this is 0.69 μs and 0.98 μs per round of syndrome extraction at $d = 25$ and $d = 31$. By this measure, SpecMatching with $k \geq 7$ dynamically keeps up with a 1 μs syndrome cycle in every configuration of Figure 8 [45]. In contrast, *pymatching* falls behind at $d = 25$ and $d = 31$ for $p = 10^{-3}$, and at $d = 31$ for $p = 5 \times 10^{-4}$. Figure 8 also shows that the gain is not limited to the mean. The whole distribution of t_{spec} lies below that of the fallback. At $k = 7$, the 99.9th percentile of the CPU components drops from 66.0 to 39.4 μs at $d = 25$, and from 128.7 to 64.9 μs at $d = 31$.

However, SpecMatching gives little or no speedup at small distances or at high noise rates. At $d = 17$ the gain is only 1.01–1.16× (Figure 8), and at $d \leq 13$ *pymatching* decodes a whole shot in 0.7–3.9 μs, much faster than SpecMatching. At $p = 3 \times 10^{-3}$, 0.6–3% of the shots fall back, and $\mathcal{H}$ percolates. In these regimes the fallback sets the latency, and the speedup is only at most 1.03×.

Divide and conquer. Figure 8 also compares SpecMatching with and without divide and conquer, using the ablation of Section 6.1.1. Without it, the largest component of $\mathcal{H}$ bounds the latency (Section 3.3), so adding solver cores does not help. At $d = 31$ and $p = 10^{-3}$, the mean stays at 41.4–42.7 μs for $k = 4, 7, 10$. Cutting and fusing the large components brings the mean down to 36.4, 30.5 and 29.8 μs for the same k. At $k = 7$, divide and conquer also lowers the 99.9th percentile from 130.6 to 67.2 μs at $d = 31$. At $d = 25$, it lowers the 99.9th percentile from 58.5 to 40.2 μs and the mean from 21.8 to 17.1 μs. But when the components of $\mathcal{H}$ have not percolated (Figure 2(c)), divide and conquer costs more than it saves. At $p = 5 \times 10^{-4}$ it is 10–20% slower on the mean, e.g. 7.9 against 7.2 μs at $d = 25$ and $k = 7$. This is because its manager does more serial work. In summary, cutting $\mathcal{H}$ helps only when the defects form large connected components that cannot be divided otherwise, and it slows the decoder down when they do not. At the current error rates of $\sim 10^{-3}$, percolation is a problem, but it may not be in the future as qubits and gates become more accurate.

Latency breakdown. To see where the time goes, Figure 9 splits t_{spec} into its stages. At $p = 5 \times 10^{-4}$, the serial work of the manager (Section 4.2.1) dominates. It accounts for 6.4 of the 7.9 μs at $d = 25$, and for 12.5 of the 14.1 μs at $d = 31$. At $p = 10^{-3}$, the solver cores carry a larger share. At $d = 31$, 13.1 μs of manager work is followed by 11.0 μs of leaf builds and solves, 2.3 μs of fusions and 2.3 μs of extraction. The preprocessor contributes only 0.1–1.7 μs, which is 2–6% of the critical path.

[1]The 0.7 V device point is a first-order scaling of the measured 0.45 V PCACTI model.

Table 2: Mean latency in μs of *pymatching* on one core, of Fusion Blossom on one core ($P = 1$) and on 9 simulated cores with the best number of time partitions P, and of SpecMatching on 9 cores ($k = 7$, preprocessor included). The last column is the ratio of the means of Fusion Blossom and SpecMatching on 9 cores. 10^5 shots per entry for Fusion Blossom.

p	d	*pymatching*	Fusion Blossom		SpecMatching	ratio
		1 core	1 core	9 cores (best P)	9 cores	
5×10^{-4}	17	3.9	69	40 ($P = 4$)	**3.7**	10.8×
	25	16.5	249	106 ($P = 4$)	**7.9**	13.4×
	31	35.7	550	198 ($P = 8$)	**14.1**	14.1×
10^{-3}	17	8.3	145	80 ($P = 4$)	**7.3**	10.9×
	25	35.0	604	241 ($P = 4$)	**17.1**	14.1×
	31	76.5	1346	473 ($P = 8$)	**30.5**	15.5×
3×10^{-3}	17	34.4	654	391 ($P = 4$)	**34.4**	11.4×
	25	132.6	3065	1394 ($P = 4$)	**132.6**	10.5×
	31	293.5	7987	3051 ($P = 4$)	**288.2**	10.6×

Preprocessor. The preprocessor is what keeps the construction of $\mathcal{H}$ such a small part of the critical path. At $p = 10^{-3}$ and $L = 64$ lanes, it takes 766 cycles on average at $d = 25$ (0.80 μs at 0.96 GHz) and 1485 cycles at $d = 31$ (1.73 μs at 0.86 GHz). This is 5–6% of the mean system latency, and 114× less than the 91.0 μs that the same construction takes on one CPU core (Section 3.2). Pruning and the time-shift layout (Section 4.1.1) halve the number of chunks that are fetched. As a result, the preprocessor is 1.40–1.57× faster than with one full ball per detector for $p \leq 10^{-3}$. At $p = 3 \times 10^{-3}$ the gain is only 1.19×, because the output port emits one edge per cycle, and this bounds the latency.

6.3 Comparison with Other Decoders

We now compare it with two exact parallel MWPM decoders, Fusion Blossom and Micro Blossom, under the setup of Section 6.1.

Fusion Blossom. Table 2 compares SpecMatching with Fusion Blossom [67] on the same nine cores. In every configuration, SpecMatching is 10.5–15.5× faster on average. Fusion Blossom on nine cores is still 5.5–11× slower than *pymatching* on one core. The best P is 4 or 8, so at most four or eight of the nine cores work at the same time. SpecMatching, in contrast, divides only the components of $\mathcal{H}$ that need it, and its fusions touch only the defects near a cut. For reference, Fusion Blossom reaches real-time throughput up to $d = 33$ at $p = 10^{-3}$ on 128 cores [67], which is 14 times more.

Micro Blossom. Micro Blossom [66] reaches 0.8 μs on average at $d = 13$ and $p = 10^{-3}$. However, $d = 15$ already occupies 867k of the 900k LUTs of its Xilinx VMK180, so the distances that we evaluate do not fit on the device. We therefore use the three-part estimate of Section 6.1.2. Its measured CPU part alone takes 0.15–0.93 μs per shot (d rounds) for $d \in \{17, 25, 31\}$ and $p \leq 10^{-3}$. With an RTT of 0.2 μs, close to the value measured for its FPGA interface, the estimated mean is 0.75–9.5 μs, which is 3.2–5.6× below SpecMatching. So for a single memory experiment, Micro Blossom has a lower predicted latency than SpecMatching.

This advantage comes with three caveats. First, the cost of logic on its accelerator array grows with $\mathcal{G}$, as we show in Section 6.5.

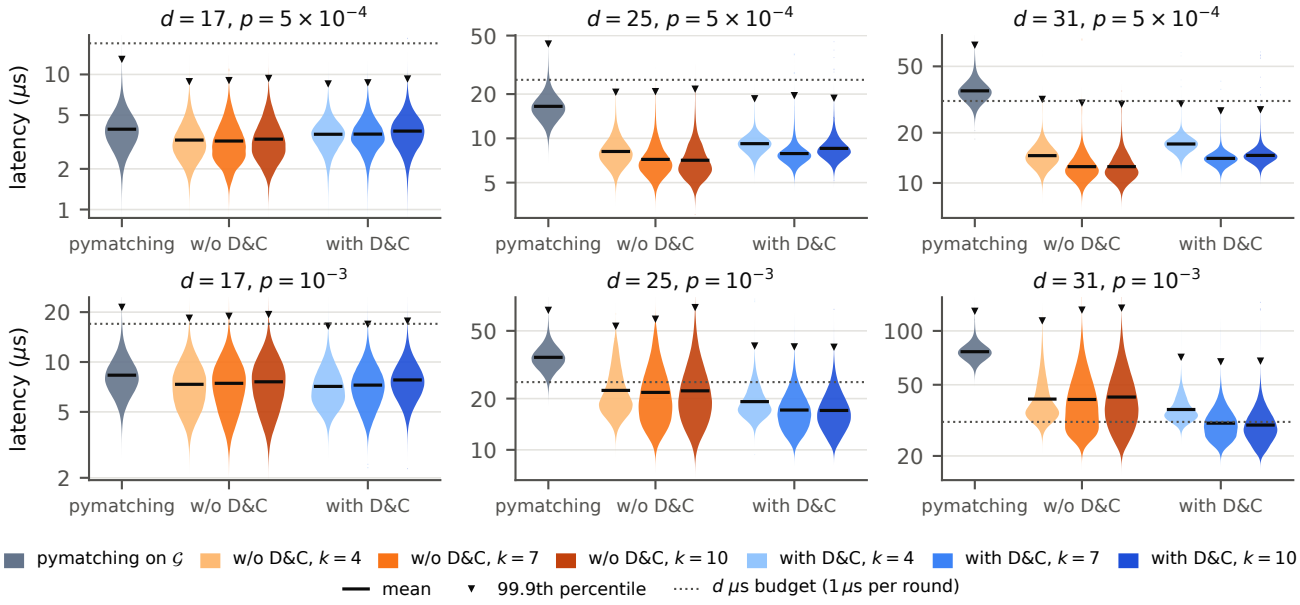


Figure 8: Per-shot latency of *pymatching* on $\mathcal{G}$ and of SpecMatching without and with divide and conquer (D&C) on $k = 4, 7, 10$ solver cores (5×10^5 shots each, log scale, preprocessor included). Without D&C, whole components of $\mathcal{H}$ are balanced over the k cores. Horizontal bars mark the mean and triangles the 99.9th percentile. Dotted lines mark the d μs budget.

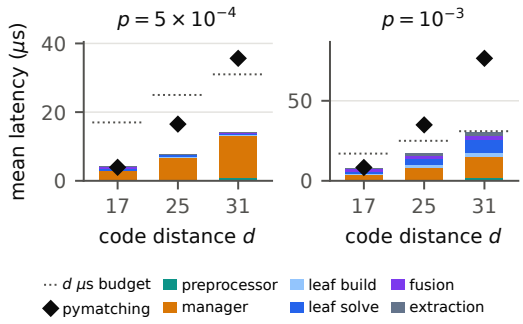


Figure 9: Critical path of t_{spec} at $k = 7$, averaged over the shots. *pymatching*'s latencies only account for the time to run sparse blossom; equivalent to leaf solve+fusion+extraction.

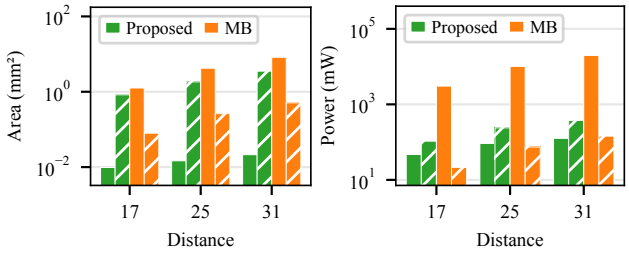
Table 3: Lattice surgery on a $1 \times N$ strip at $d = 25$, $k = 7$, $T = 2$, $W = 2$: mean latency in μs of SpecMatching on $N(k + 2) + 1$ cores, preprocessors included (per round in parentheses), of *pymatching* on the whole strip on one core, the speedup over *pymatching* on the strip; and the fraction of shots in which some patch falls back. 10^5 shots per entry.

p	N	SpecMatching	*pymatching*	speedup	fallback rate
5×10^{-4}	2	8.3 (0.33)	40.0	4.8×	4.0×10^{-5}
	4	10.5 (0.42)	107.7	10.3×	1.0×10^{-4}
	8	13.6 (0.54)	257.5	19.0×	1.9×10^{-4}
	16	15.2 (0.61)	505.7	33.3×	6.0×10^{-4}
10^{-3}	2	21.1 (0.85)	89.4	4.2×	3.4×10^{-4}
	4	30.9 (1.23)	202.6	6.6×	9.0×10^{-4}
	8	45.1 (1.81)	526.4	11.7×	1.9×10^{-3}
	16	47.4 (1.90)	802.0	16.9×	4.0×10^{-3}

Second, it hard-codes 4-bit edge weights. These keep only 3 of the 31 distinct weights of our noise model, and merge edges whose error probabilities differ by up to 3×. Yet edge weights are what raise the threshold of MWPM under circuit-level noise [36]. Third, one Micro Blossom instance cannot be split across space. All of its processing units grow their regions in lockstep, by a step length that is the minimum over the whole array. Two patches decoded on two instances have to be joined by exchanging this global minimum and the state of the units on the seam on every instruction. Lattice surgery therefore needs either one array for the whole merged region, or a decomposition such as modular decoding [6], which re-decodes a buffer of width d around every seam. SpecMatching, instead, fuses adjacent patches exactly, once per shot, as the next experiment shows.

6.4 Lattice Surgery Experiment

We decode the lattice-surgery strip described in Section 6.1.3 using N SpecMatching instances. Table 3 shows the mean latency of decoding the strip at $d = 25$ and $k = 7$, and compares it with *pymatching* on the whole strip. The latency of *pymatching* grows faster than linearly with N. In contrast, the latency of SpecMatching grows slowly, because every patch is solved in parallel and only the seams between patches are fused. As a result, the speedup



(a) Area vs distance. The left bar and right bar (hatched) are logic and SRAM area.

(b) Power vs distance. The left bar and right bar (hatched) are dynamic and leakage power.

Figure 10: Hardware for preprocessor at $p = 10^{-3}$ and $L = 64$. MB stands for Micro Blossom (16 bit edge weights).

grows from 4.2–4.8× for $N = 2$ patches to 17–33× for $N = 16$. The preprocessors add only 0.34 μs at $p = 5 \times 10^{-4}$ and 0.80 μs at $p = 10^{-3}$, because every patch builds its part of $\mathcal{H}$ in parallel.

At $p = 5 \times 10^{-4}$, the strip stays below 1 μs per round up to $N = 16$. At $p = 10^{-3}$, however, it exceeds 1 μs per round from $N = 4$. We attribute this to large components of $\mathcal{H}$ that span a whole patch. Patch i lies between seams $i-1$ and i. If no component of $\mathcal{H}_i$ touches both seams, the fusions at the two seams can run in parallel. Components that span the patch become more common at higher p, and the seams then cost more. At $p = 10^{-3}$ and $N = 8$, such components occur in 95% of the shots, and the seam fusions take 2.4 sequential fusion jobs (Section 4.2.2) on average. This is twice the 1.04 jobs at $p = 5 \times 10^{-4}$. Moreover, the extraction jobs on the large fused roots take a third of the critical path.

Shots in which some patch falls back raise the mean further. At $d = 25$, a patch falls back at roughly N times the rate of a single patch. The resolution of such a fallback stays local. The territory Ω (Section 5), the region that is decoded at the full resolution of $\mathcal{G}$, touches 2.0–3.5 patches on average, and at most 7. In the current implementation, however, phase 2, which joins the fallback state with the speculative state of the other patches, is slow on these shots. At $d = 25$ it takes 0.2–1.7 ms on average, more than the 0.05–0.8 ms of *pymatching* on the whole strip. This is because, in nearly every such shot, Ω grows beyond the slab that the fallback core preloads. Despite this, SpecMatching is better because its mean latency is much smaller.

6.5 Accelerator Area and Power

We now present what the preprocessor costs in silicon, and compare it with Micro Blossom's accelerator (Section 6.3).

Area. Figure 10a compares the area of the two designs across code distances. The two designs are built in opposite ways. Micro Blossom is 94.19% replicated logic, and its vertex and edge memories only take 5.81% of its area at $d = 31$. Our preprocessor, in contrast, is 99.40% SRAM, dominated by its ball table. At $d = 31$, Micro Blossom occupies 8.77 mm^2 and our preprocessor 3.54 mm^2, 2.48× less. Across $d = 17$, 25 and 31, the area of our preprocessor is 0.63×, 0.41× and 0.40× that of Micro Blossom. This is because Micro Blossom must replicate a processing unit for every vertex of $\mathcal{G}$. The compute of SpecMatching is instead sized by the number of defects per shot, and its dependence on d is confined to memory, which is denser and cheaper than logic.

Power Figure 10b compares the power of the two designs. Micro Blossom's power is 99.28% dynamic: at $d = 31$ it consumes 19.76 W over its 282 ns round at 1 GHz, with only 144.17 mW of leakage. Our preprocessor consumes total of 501.44 mW at $d = 31$. Across $d = 17$, 25 and 31, Micro Blossom consumes 20.06×, 30.85× and 39.70× more power. As with its area (99.40%), the power of our preprocessor is dominated by SRAM (82.03% at $d = 31$). This is 375.42 mW of table leakage, which is spent regardless of activity, plus 35.90 mW of table reads, while the remaining logic draws only 90.06 mW.

7 Related Work

Accuracy vs latency. Several prior works attempt to solve matching suboptimally by trading off accuracy for latency. Cellular automaton [56] and neural [8, 9] predecoders, the lazy decoder [13], Clique [49], Astrea-G and Promatch [2, 63] and [64] resolve the easy part of a syndrome locally, and union-find decoders [3, 12, 15, 30, 36, 44] grow and merge clusters around the defects. All of them rely on the fact that below threshold, the defects are sparse and clustered. But their local decisions are heuristics without the guarantees of an exact global matching. SpecMatching promotes these locality and clustering ideas into an exact certification mechanism: it runs exact blossom inside every clustered component of $\mathcal{H}$, and a dual certificate decides whether the local view suffices.

Decoding frameworks. Prior works also combine multiple decoders to decode a large problem in several ways. Window decoding splits the stream of syndromes into overlapping pieces. Sliding [16], parallel [55, 58] and modular [6] window decoding are agnostic to the inner decoder, which SpecMatching can serve as. Alternatively, SpecMatching's seam fusion joins windows exactly, without a buffer. In either case, global coordination between adjacent syndrome blocks is required. FusionQ [70] decodes qLDPC codes that need not be matchable using a clustering and fusion technique similar to ours. SWIPER [62] speculates on the corrections passed across window seams [41], whereas SpecMatching speculates only on whether $\mathcal{H}$ contains the optimal matching. Another closely related idea is decoder switching [61]. It invokes a slow, strong decoder when the output of a fast, weak decoder is unreliable. SpecMatching instead switches on an exact certificate, and runs both decoders concurrently. Decoder scheduling [46] allocates decoders across a program, complementary to SpecMatching's allocation of cores within a syndrome block.

8 Conclusion and Future Work

SpecMatching is an exact MWPM decoder that accelerates decoding by speculation and parallelization. It speculates that the optimal matching is contained in a sparse graph $\mathcal{H}$ which it builds using a dedicated hardware accelerator. Compared to *pymatching*, it reduces the mean latency by 2× at $d = 25$ and 2.5× at $d = 31$ under a noise rate of $p = 10^{-3}$. It is also *composable*, meaning multiple SpecMatching instances can be combined to solve decoding problems that span space or time. There are several promising future directions. (a) Further accelerating SpecMatching by parallelizing the manager's serial work on a preprocessor [3, 12, 40]. (b) A faster local

fallback for lattice surgery. The fallback cores can be re-purposed to supply edges longer than $2T$ to the manager core by running multi-source Dijkstra. Edges *can* be added to $\mathcal{H}$ mid blossom, provided they have non-zero slack $w_e - \sum_{S:e\in\delta(S)} y_S \geq 0$. (c) Extending dual truncation and speculation to modular matchable codes [34] and color code [26, 51]. (d) Understanding the theory of percolation phases (Section 3.3) and thresholds in QEC decoding problems [70].

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number No.DE-SC0026525. Claude Opus 5.5 (by Anthropic) was used extensively to write the code, run the experiments and help write and edit the manuscript. We thank Austin Fowler and the TQEC community [57] for helpful discussions.

References

[1] Dorit Aharonov and Michael Ben-Or. 1997. Fault-tolerant quantum computation with constant error. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. 176–188. doi:10.1145/258533.258579

[2] Narges Alavisamani, Suhas Vittal, Ramin Ayanzadeh, Poulami Das, and Moinuddin Qureshi. 2024. Promatch: Extending the reach of real-time quantum error correction with adaptive predecoding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 818–833.

[3] Ben Barber, Kenton M. Barnes, Tomasz Bialas, Okan Buğdaycı, Earl T. Campbell, Neil I. Gillespie, Kauser Johar, Ram Rajan, Adam W. Richardson, Luka Skoric, Canberk Topal, Mark L. Turner, and Abbas B. Ziad. 2025. A real-time, scalable, fast and resource-efficient decoder for a quantum computer. *Nature Electronics* 8, 1 (2025), 84–91. doi:10.1038/s41928-024-01319-5

[4] Francesco Battistel, Christopher Chamberland, Kauser Johar, Ramon W. J. Overwater, Fabio Sebastiano, Luka Skoric, Yosuke Ueno, and Muhammad Usman. 2023. Real-time decoding for fault-tolerant quantum computing: progress, challenges and outlook. *Nano Futures* 7, 3 (2023), 032003. doi:10.1088/2399-1984/aceba6

[5] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. 2022. Assessing requirements to scale to practical quantum advantage. *arXiv preprint arXiv:2211.07629* (2022).

[6] Héctor Bombín, Chris Dawson, Ye-Hua Liu, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. 2023. Modular decoding: parallelizable real-time decoding for quantum computers. *arXiv preprint arXiv:2303.04846* (2023).

[7] Todd A Brun. 2019. Quantum error correction. *arXiv preprint arXiv:1910.03672* (2019).

[8] Christopher Chamberland, Luis Goncalves, Prasahnt Sivarajah, Eric Peterson, and Sebastian Grimberg. 2023. Techniques for combining fast local decoders with global decoders under circuit-level noise. *Quantum Science and Technology* 8, 4 (2023), 045011. doi:10.1088/2058-9565/ace64d

[9] Christopher Chamberland, Jan Olle, Muyuan Li, Scott Thornton, and Igor Baratta. 2026. Fast and accurate AI-based pre-decoders for surface codes. *arXiv preprint arXiv:2604.12841* (2026).

[10] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm FinFET Predictive Process Design Kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006

[11] George B. Dantzig. 1991. *Linear Programming and Extensions.* Princeton University Press. http://www.jstor.org/stable/j.ctt1cx3tvg

[12] Poulami Das, Christopher A. Pattison, Srilatha Manne, Douglas M. Carmean, Krysta M. Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2022. AFS: Accurate, Fast, and Scalable Error-Decoding for Fault-Tolerant Quantum Computers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 259–273. doi:10.1109/HPCA53966.2022.00027

[13] Nicolas Delfosse. 2020. Hierarchical decoding to reduce hardware requirements for quantum computing. *arXiv preprint arXiv:2001.11427* (2020).

[14] Nicolas Delfosse, Vivien Londe, and Michael E Beverland. 2022. Toward a union-find decoder for quantum LDPC codes. *IEEE Transactions on Information Theory* 68, 5 (2022), 3187–3199.

[15] Nicolas Delfosse and Naomi H. Nickerson. 2021. Almost-linear time decoding algorithm for topological codes. *Quantum* 5 (Dec. 2021), 595. doi:10.22331/q-2021-12-02-595

[16] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (2002), 4452–4505.

[17] Peter-Jan HS Derks, Alex Townsend-Teague, Ansgar G Burchards, and Jens Eisert. 2025. Designing fault-tolerant circuits using detector error models. *Quantum* 9 (2025), 1905.

[18] Edsger W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1 (1959), 269–271. doi:10.1007/BF01386390

[19] Austin G. Fowler. 2012. Proof of Finite Surface Code Threshold for Matching. *Phys. Rev. Lett.* 109 (Nov 2012), 180502. Issue 18. doi:10.1103/PhysRevLett.109.180502

[20] Austin G. Fowler. 2015. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average O(1) parallel time. *Quantum Info. Comput.* 15, 1–2 (Jan. 2015), 145–158.

[21] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics* 86, 3 (2012), 032324.

[22] Iulia M Georgescu, Sahel Ashhab, and Franco Nori. 2014. Quantum simulation. *Reviews of Modern Physics* 86, 1 (2014), 153–185.

[23] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.

[24] Craig Gidney. 2025. How to factor 2048 bit RSA integers with less than a million noisy qubits. *arXiv preprint arXiv:2505.15917* (2025).

[25] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433. doi:10.22331/q-2021-04-15-433

[26] Craig Gidney and Cody Jones. 2023. New circuits and an open source decoder for the color code. *arXiv preprint arXiv:2312.08813* (2023).

[27] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681. doi:10.1038/s41586-022-05434-1

[28] Google Quantum AI. 2025. Quantum error correction below the surface code threshold. *Nature* 638 (2025), 920–926. arXiv:2408.13687 [quant-ph] doi:10.1038/s41586-024-08449-y

[29] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction.* California Institute of Technology.

[30] Sam J. Griffiths and Dan E. Browne. 2024. Union-find quantum decoding without union-find. *Physical Review Research* 6, 1 (2024), 013154. doi:10.1103/PhysRevResearch.6.013154

[31] Andi Gu, J Ataides, Mikhail D Lukin, and Susanne F Yelin. 2026. Scalable neural decoders for practical fault-tolerant quantum computation. *arXiv preprint arXiv:2604.08358* (2026).

[32] Oscar Higgott, Thomas C. Bohdanowicz, Aleksander Kubica, Steven T. Flammia, and Earl T. Campbell. 2023. Improved Decoding of Circuit Noise and Fragile Boundaries of Tailored Surface Codes. *Physical Review X* 13, 3 (2023), 031007. doi:10.1103/PhysRevX.13.031007

[33] Oscar Higgott and Craig Gidney. 2025. Sparse blossom: correcting a million errors per core second with minimum-weight matching. *Quantum* 9 (2025), 1600.

[34] Oscar Higgott, Hasan Sayginel, Francisco JH Heras, Zhiyang He, Tomas Jochym-O'Connor, Andrew W Senior, Lei M Zhang, Thomas Edlich, James S Spencer, Matt McEwen, Craig Gidney, Johannes Bausch, Pushmeet Kohli, and Hartmut Neven. 2026. Low-Overhead Quantum Error Correction with Boundary-Connected Planar Modules. *arXiv preprint arXiv:2610.03682* (2026).

[35] Dominic Horsman, Austin G. Fowler, Simon Devitt, and Rodney Van Meter. 2012. Surface code quantum computing by lattice surgery. *New Journal of Physics* 14, 12 (2012), 123011. doi:10.1088/1367-2630/14/12/123011

[36] Shilin Huang, Michael Newman, and Kenneth R. Brown. 2020. Fault-tolerant weighted union-find decoding on the toric code. *Physical Review A* 102, 1 (2020), 012419. doi:10.1103/PhysRevA.102.012419

[37] A Yu Kitaev. 2003. Fault-tolerant quantum computation by anyons. *Annals of physics* 303, 1 (2003), 2–30.

[38] Jon Kleinberg and Éva Tardos. 2006. *Algorithm Design.* Addison Wesley.

[39] Emanuel Knill, Raymond Laflamme, and Wojciech H. Zurek. 1998. Resilient Quantum Computation. *Science* 279, 5349 (1998), 342–345. doi:10.1126/science.279.5349.342

[40] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1990. Efficient parallel algorithms for graph problems. *Algorithmica* 5, 1 (1990), 43–64.

[41] Jocelyn Li and Margaret Martonosi. 2026. An Analysis of Speculative Window Decoders for Quantum Error Correction. *arXiv preprint arXiv:2606.24048* (2026).

[42] Shuang Liang, Jubo Xu, Giulio Bassanino, Qianzhou Wang, Yidong Zhou, Yuncheng Lu, Zhiwen Mo, Paul Kelly, Bo Yuan, Wayne Luk, and Hongxiang Fan. 2026. Coset Ensemble Decoder for Quantum Error Correction with Algorithm-Hardware Co-Design. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 857–871.

[43] Ye-Hua Liu and David Poulin. 2019. Neural belief-propagation decoders for quantum error-correcting codes. *Physical review letters* 122, 20 (2019), 200501.

[44] Namitha Liyanage, Yue Wu, Siona Tagare, and Lin Zhong. 2024. FPGA-based distributed union-find decoder for surface codes. *IEEE Transactions on Quantum Engineering* 5 (2024), 1–18.

[45] Robert M. Loynes. 1962. The stability of a queue with non-independent inter-arrival and service times. *Mathematical Proceedings of the Cambridge Philosophical Society* 58, 3 (1962), 497–520.

[46] Satvik Maurya, Abtin Molavi, Aws Albarghouthi, and Swamit Tannu. 2026. A Case for Elastic Quantum Error Correction Decoders. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. arXiv:2406.17995.

[47] Anthony Ryan O'Rourke and Simon Devitt. 2025. Compare the pair: Rotated versus unrotated surface codes at equal logical error rates. *Physical Review Research* 7, 3 (2025), 033074.

[48] Pavel Panteleev and Gleb Kalachev. 2021. Degenerate quantum LDPC codes with good finite length performance. *Quantum* 5 (2021), 585.

[49] Gokul Subramanian Ravi, Jonathan M Baker, Arash Fayyazi, Sophia Fuhui Lin, Ali Javadi-Abhari, Massoud Pedram, and Frederic T Chong. 2023. Better than worst-case decoding for quantum error correction. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[50] Joschka Roffe. 2019. Quantum error correction: an introductory guide. *Contemporary Physics* 60, 3 (2019), 226–245.

[51] Kaavya Sahay and Benjamin J Brown. 2022. Decoder for the triangular color code by matching on a möbius strip. *PRX Quantum* 3, 1 (2022), 010310.

[52] Sumeet Shirgure. 2026. *SpecMatching: Accelerating PyMatching with Truncated Sparse Blossom and Graph Sparsification.* https://github.com/sumeetshirgure/SpecMatching GitHub repository.

[53] Peter W. Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Physical Review A* 52, 4 (1995), R2493–R2496. doi:10.1103/PhysRevA.52.R2493

[54] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509. doi:10.1137/S0097539795293172

[55] Luka Skoric, Dan E Browne, Kenton M Barnes, Neil I Gillespie, and Earl T Campbell. 2023. Parallel window decoding enables scalable fault tolerant quantum computation. *Nature Communications* 14, 1 (2023), 7040. doi:10.1038/s41467-023-42482-1

[56] Samuel C Smith, Benjamin J Brown, and Stephen D Bartlett. 2023. Local predecoder to reduce the bandwidth and latency of quantum error correction. *Physical Review Applied* 19, 3 (2023), 034050.

[57] Adrien Suau, Yiming Zhang, Purva Thakre, Yilun Zhao, Kabir Dubey, Jose A. Bolanos, Arabella Schelpe, Tianyi Hao, Philip Seitz, Gian Giacomo Guerreschi, Ángela Elisa Álvarez Pérez, Reinhard Stahn, Jerome Lenssen, Brendan Reid, and Austin Fowler. 2026. tqec: A Python package for topological quantum error correction. *Journal of Open Source Software* 11, 120 (2026), 9142. doi:10.21105/joss.09142

[58] Xinyu Tan, Fang Zhang, Rui Chao, Yaoyun Shi, and Jianxin Chen. 2023. Scalable surface-code decoders with parallelization in time. *PRX Quantum* 4, 4 (2023), 040344.

[59] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. doi:10.1145/321879.321884

[60] Barbara M. Terhal. 2015. Quantum error correction for quantum memories. *Reviews of Modern Physics* 87, 2 (2015), 307–346. doi:10.1103/RevModPhys.87.307

[61] Riki Toshio, Kaito Kishi, Jun Fujisaki, Hirotaka Oshima, Shintaro Sato, and Keisuke Fujii. 2025. Decoder Switching: Breaking the Speed-Accuracy Tradeoff in Real-Time Quantum Error Correction. *arXiv preprint arXiv:2510.25222* (2025).

[62] Joshua Viszlai, Jason D Chadwick, Sarang Joshi, Gokul Subramanian Ravi, Yanjing Li, and Frederic T Chong. 2024. Predictive Window Decoding for Fault-Tolerant Quantum Programs. *arXiv preprint arXiv:2412.05115* (2024).

[63] Suhas Vittal, Poulami Das, and Moinuddin Qureshi. 2023. Astrea: Accurate Quantum Error-Decoding via Practical Minimum-Weight Perfect-Matching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 16 pages. doi:10.1145/3579371.3589037

[64] Peter Wegmann, Theofilos Augoustis, Aleksandra Ślwierkowska, Emmanouil Giortamis, and Pramod Bhatotia. 2026. Zero-G: a pre-decoder-aware decoder for quantum error correction. *arXiv preprint arXiv:2608.02030* (2026).

[65] Yue Wu, Namitha Liyanage, and Lin Zhong. 2022. An interpretation of Union-Find Decoder on Weighted Graphs. *arXiv preprint arXiv:2211.03288* (2022).

[66] Yue Wu, Namitha Liyanage, and Lin Zhong. 2025. Micro blossom: Accelerated minimum-weight perfect matching decoding for quantum error correction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 639–654.

[67] Yue Wu and Lin Zhong. 2023. Fusion blossom: Fast mwpm decoders for qec. In *2023 IEEE international conference on quantum computing and engineering (QCE)*, Vol. 1. IEEE, 928–938.

[68] Satoshi Yoshida, Ethan Lake, and Hayata Yamasaki. 2026. Proof of a finite threshold for the union-find decoder. *arXiv preprint arXiv:2602.20238* (2026).

[69] Boqing Zhang, Henry D Pfister, Hanwen Yao, and Siyuan Niu. 2026. Towards Scalable Quaternary Message-Passing Decoding for Quantum Error Correction. *arXiv preprint arXiv:2605.24177* (2026).

[70] Junyu Zhou, Sean Garner, Aarthi Sundaram, Gushu Li, and Yue Wu. 2026. FusionQ: A Generic and Optimal Decoding Framework for Ultra-Low-Latency Quantum Error Correction. arXiv:2611.13111 [quant-ph] https://arxiv.org/abs/2611.13111