QuErK: Accelerating Quantum Error Correction through FPGAs

Valentino Guerrini, Marco Venere,
Pietro Giannoccaro, Beatrice Branchini

June 30, 2023

Team Number: xohw23-251
GitHub Repo:
`https://github.com/PietroG51/xohw23-Accelerating-QEC-FPGAs`
YouTube Video: `https://youtu.be/GexVrHjFZVw`

Abstract

Computational power demand is constantly growing. The dawn of Moore's Law has already happened in 2021, a decade sooner than predicted in 2013 [16] thus introducing a gap between the increasing computational demand and current hardware architectures capability causing huge research efforts towards new computation paradigms [2]. Quantum Computing is one of the most promising fields of research for the future of computation as it exploits the unique properties of quantum physics to reach unseen levels of parallelism and solve specific problems that are practically not solvable by classical computers. One of the most important challenges to be faced is unacceptable error rates due to environmental noise which leads to quantum decoherence. An error correction system is therefore needed to reach higher reliability and fault-tolerant architectures. Quantum Error Correction techniques are employed to encode the logical qubits in many more physical qubits, with some of them being periodically read to gain information about occurred errors. The obtained outcomes are then delivered to a decoder running on an external classical computing device which provides the most probable error pattern that may have occurred, in order to provide back to the quantum system a correction. The whole cycle is called *round of correction* and must be done within $1\mu s$ to obtain real-time QEC. This work proposes **QuErK** (**Qu**antum **Er**ror Correction **K**ernel), which pursues an FPGA-based approach for the acceleration of an online decoder with Sparse Blossom[10] decoding algorithm to drastically reduce error rates in quantum systems. This is done by accelerating one function belonging to the graph flooder, called *find_next_event*, which computes the collision times of pairs of regions, in order to find the most upcoming event to handle. Our solution is the first step towards the full implementation of the Sparse Blossom on FPGA, and by using a host code coupled with a *AMD-Xilinx Alveo U55C*[21] is able to provide 49x with respect to the software Python version executed on setup with *AMD Ryzen 7 3700X* and 32 *GB of RAM*. We also show that the hardware accelerator we designed is about $50x$ faster than the software version, and $118x$ more energy efficient, thus proving to approach the requirements given by a quantum hardware management system.

1 Introduction

Quantum Computing is one of the most promising fields of research for the future of computation to revolutionize many applications, including communications, AI and machine learning, and cybersecurity. One of the most important challenges to be faced is unacceptable error rates due to the susceptibility of the system to environmental noise which leads to quantum decoherence. An error correction system is therefore needed to reach higher reliability of the quantum computer as well as a fault-tolerant architecture. Quantum Error Correction techniques rely on encoding logical qubits in many more physical qubits, which are read periodically through a syndrome extraction circuit while the system is running. The syndrome is then delivered to a decoder running on an external classical computing device which provides the most probable error that may have occurred to provide a correction back to the quantum system.

This work pursues an FPGA-based approach for the Sparse Blossom[10] decoding algorithm acceleration to reach a real-time error correction.

In particular, we implement a dataflow architecture that tackles the acceleration of the bottleneck of the algorithm, with a speedup of about $50\times$ with respect to the software version.

We tested our solution on AMD-Xilinx Alveo U55C, to validate the overall performance.

In Section 2, we provide the required background to better understand the bases of our work. In Section 3, we describe the algorithm. In Section 4, we present the overall architecture that we designed; finally, in Section 5, we provide our experimental results.

2 Background

Computational power demand is constantly growing as well as the increasing impact of electronics and computing devices. Artificial intelligence and cybersecurity [7] are just some of the applications with a high computational impact that are spreading.

Furthermore, the dawn of Moore's Law has already happened in 2021, a decade sooner than predicted in 2013 [16]. The saturation of performance improvement factor due to transistor scaling is introducing a gap between the increasing computational demand and current hardware architectures causing huge research efforts towards new computation paradigms [2].

2.1 A New Paradigm: Quantum Computing

Quantum computing promise to solve specific problems that are practically not solvable by classical devices in an acceptable amount of time. It is a completely new paradigm that relies on unique properties of quantum physics such as superposition and entanglement of states to reach unseen levels of

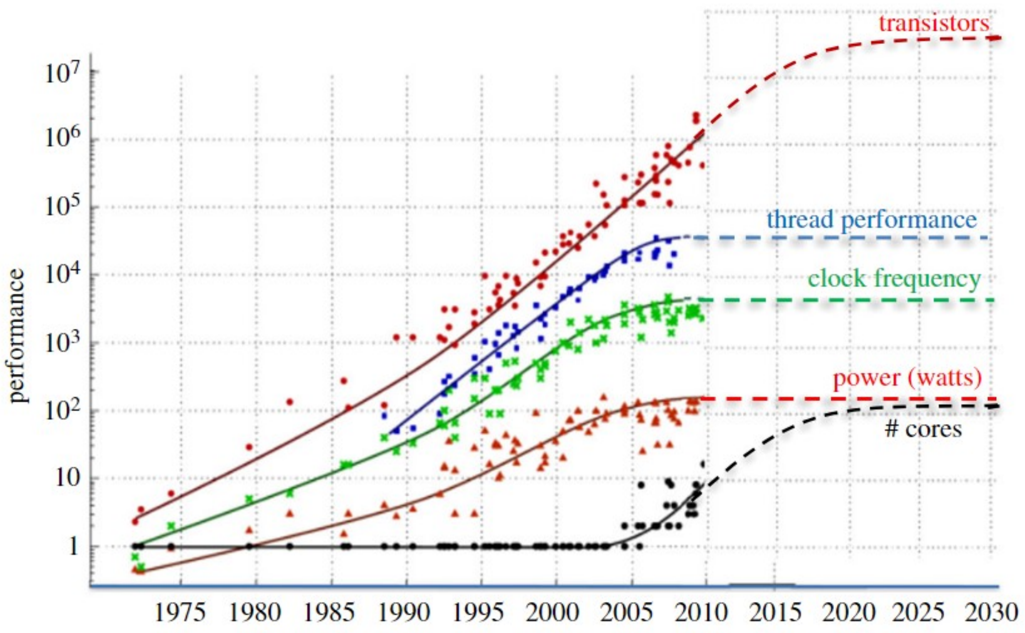


Figure 1: The dawn of Moore's Law [16]. Performance improvement with transistors scaling is coming to an end.

parallelism [7]. There has been a huge effort from the research community to reach "quantum supremacy", which is the practical demonstration of the ability of quantum devices to solve problems that cannot be handled by classical computers.

2.2 Demonstrating Quantum Supremacy: Challenges To Be Faced

As of today, there is no proof that a large fault-tolerant quantum computer can be effectively realized [2] as well as the demonstration of quantum supremacy since there are many challenges to be faced. First of all, there are many technological approaches for qubits realization such as trapped ions, silicon-based spin, and cold atom technology; each of them with its pros and cons on size, coherence time, operational temperature, scalability, and read-out fidelity [7]. The realization of a quantum computer is very complex and expensive and involves many fields, from Computer Science to Physics and Material Science [2].

Moreover, many classical computing paradigms cannot be applied to quantum computing. To make an example, quantum states cannot be copied due to the no-cloning theorem. Also, reading the state of a qubit make it collapse and lose its quantum properties such as the quantum superposition of states.

In conclusion, noise has a notable impact on the reliability of the quantum computer. Fabrication imperfections, crosstalk, sensitivity to parasitic signals, and material degradation are just some of the environmental factors [7] that can lead to quantum decoherence, causing loss of information and errors due to the modification of qubits states. This lead to high error rates of quantum devices, in the order of 1% [1] whereas the target applications require an error about 10^{-10} [4].

2.3 Quantum Error Correction

One of the most promising techniques to reach lower error rates and fault-tolerant quantum computing is Quantum Error Correction. The basic concept behind quantum error correction consists of encoding data qubits in many more physical qubits [9]. The goal is to exploit the redundancy of information and the larger the number of physical qubits the lower the error rates. A reasonably fault-tolerant logical qubit can require several physical qubits in the order of 10^3 to 10^4 [9], which implies a large physical size and time of computation. However, it must be noted that the actual number of physical qubits strongly depends on the error model and the chosen technology.

Error correction requires architectural changes in the quantum system and the addition of a classical computing device which must support the quantum computer [4]. Regarding the quantum environment, as already mentioned, many physical qubits are needed, in the so-called *surface codes.* These can be either *data qubits* to actually store information, or *parity qubits* to acknowledge pieces of information about the occurrence of errors. On the classical computing side, a periodical read of the parity qubits, called *syndrome* extraction, must be performed. The syndrome is then forwarded to a decoder, which depending on the error model and the chosen algorithm extracts the most probable error that may have occurred. A correction is therefore computed and provided back to the quantum computer. The improvement in error rates depends on the accuracy of the chosen decoding algorithm and the performance of the implemented decoder.

Real-time error correction consists of decoding and providing corrections faster than the arrival rate of syndromes, which is in the order of one microsecond per round [10]. The one-microsecond threshold is a strict requirement that introduces an important trade-off between accuracy and speed for decoders.

2.4 Surface Codes

One of the most promising approaches for quantum error correction implementation are *surface codes* [9], which encode logical qubits into a 2-dimensional lattice alternating data and parity qubits [4]. The overall number of physical qubits is defined by the code distance d. In such structures, one parity qubit is shared between 4 data qubits called stabilizers, as shown in Figure 2. In the code, two types of errors can happen: X errors and Z errors. The X errors are bit-flips that swap the probability amplitudes of the basis states. The Z errors instead, are phase flips that introduce a relative phase of -1. The Pauli Y includes both bit and phase flip errors, X and Z. Moreover, in surface codes, the Z stabilizers data qubit are responsible for the detection of X errors whereas the X stabilizers are responsible for Z errors detection. Consequently, two sets of syndromes are obtained and forwarded to the X-decoder and the Z-decoder [4].

To give an idea of the number of qubits as a function of the distance a table reported in [4] is shown:

Table 1: Distance d and qubits in surface codes

d	**Data**	**X-Parity**	**Z-Parity**	**Total Physical**
3	9	4	4	17
4	16	8	7	31
5	25	12	12	49

2.5 The Decoding Graph

Once the syndrome is extracted, every parity qubit, also called *detector*, is read. Its outcome is 1 if the observed parity differs from the parity expected in a noiseless circuit. A *logical observable* is also defined, as a linear combination of measurement bits, whose outcome corresponds to the measurement of a logical Pauli operator [10]. Also, the error model is defined as a set of independent error mechanisms $\mathbf{e}$, each one with an assigned probability to flip detectors and observables that are all collected in the vector $\mathbf{p}$.

Additionally, two fundamental matrices are computed: the *detector check matrix* $H[i, j]$ which has on rows all the detectors and on columns all the error mechanisms; the value in the matrix is 1 if the i^{th} detector has been flipped by the j^{th} error mechanism. Similarly, the *logical observable matrix* L is defined, with the i^{th} logical observable flipped by the j^{th} error type. Indeed, each error mechanism is described by the detectors and observables it flips.

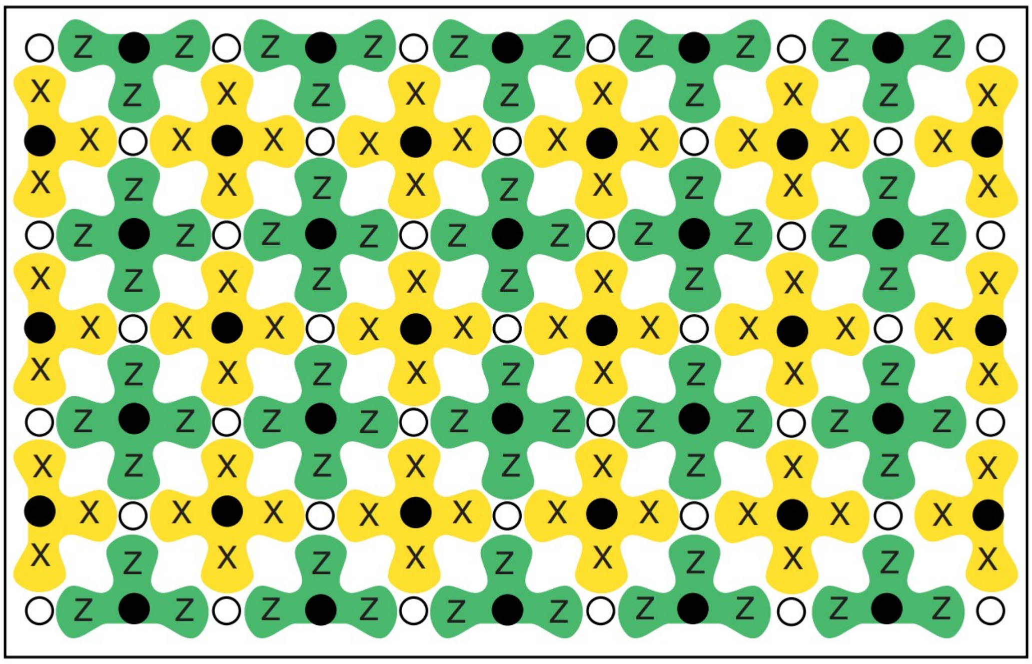


Figure 2: Architecture of surface codes [9]. In white the data qubits; in green the Z-detectors qubits; in yellow the X-detectors qubits.

In this context, the syndrome mentioned before is defined as the outcome of detector measurements given by computing $\mathbf{s} = H\mathbf{e}$ [10].

It is now possible to define the *detector graph* (also called *decoding graph*) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ made of a set of vertices $v \in \mathcal{V}$ and a set edges $e \in \mathcal{E}$. A vertex v corresponds to a triggered detector. An edge e between two vertices represents an error that has triggered the two corresponding detectors. Also, a virtual boundary node v_b is defined and sometimes there can be an edge, called a *half-edge*, linking it to a triggered detector. Each edge has a certain weight $w(e_i) \in \mathbf{w}$, which value is a function of the probability of the error associated with that edge. For example, in the Sparse Blossom algorithm[10], this function is:

$$w(e_i) = log\left[\frac{1-\mathbf{p}[i]}{\mathbf{p}[i]}\right] \tag{1}$$

Once the decoding graph has been obtained, the goal of quantum error correction is to find the most probable set of physical errors $\mathbf{c}$ consistent with the measured syndrome $\mathbf{s}$. The *minimum-weight perfect matching (MWPM)* decoder, on which the Sparse Blossom algorithm is based, computes the probability $P(\mathbf{e})$ of a set of error $\mathbf{e}$ as:

$$P(\mathbf{e}) = \prod_i (1-\mathbf{p}[i])^{(1-\mathbf{e}[i])}\mathbf{p}[i]^{\mathbf{e}[i]} = \prod_i (1-\mathbf{p}[i]) \prod_i \left(\frac{\mathbf{p}[i]}{1-\mathbf{p}[i]}\right)^{\mathbf{e}[i]} \tag{2}$$

This value is computed for all possible error sets $\mathbf{c}$ coherent with the extracted syndrome and it is chosen the one with the highest probability value. The decoding problem can be also expressed as the minimization of the function $\sum_i \mathbf{w}[i]\mathbf{c}[i]$ which corresponds to the sum of the weights of corresponding edges in $\mathcal{G}$. It must be noted that to obtain a perfect matching each node must be matched to one and only one other node included in the graph as triggered detectors. In conclusion, given a detector graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a subset of detection events $\mathcal{D} \subseteq \mathcal{V}$, it is defined a *path graph* $\overline{\mathcal{G}} = (\mathcal{D}, \overline{\mathcal{E}})$ in which every edge has a weight equal to the distance $D(u, v)$, with u, v being the vertices of an edge in $\mathcal{G}$, which corresponds to the length of the shortest path between u and v. Therefore, the problem can be also seen as graphically finding near nodes thus creating and matching between them which consists in associating an edge (error) that connects the two vertices.

2.6 Decoding Algorithms and State of the Art analysis

The *Minimum-Weight Perfect Matching* (MWPM) has worst-case complexity in the graph of $O(n^3 log(N))$, being N the number of nodes; the running time instead is roughly $O(N^2)$. Fowler [8] has proposed an MWPM implementation with $O(1)$ parallel running time. The MWPM is not the only algorithm employed for Quantum Error Correction and several approaches have been pursued. The *Union-Find* (UF) [5][6][12][20] decoder has an almost-linear worst-case running time as N increases but lacks in terms of accuracy [10]. Other approaches are *Maximum-likelihood* decoders [3][17] which can provide higher accuracy at the cost of increasing execution times, and recently *Neural Networks* decoders [14][15][13] which try to improve performance on accuracy while reducing the timing impact.

This work is based on *PyMatching2* [10], an open-source implementation of the *Sparse Blossom* algorithm, which solves the MWPM decoding problem by acting directly on the decoding graph. The results reported by [10] are very promising in terms of timing, fundamental to achieving a real-time error correction within the one-microsecond time window constraint. As it is reported, a $d = 29$ distance code (which corresponds to a 10^{-12} logical error rate) takes about $3.5\mu s$ per round on a single core, therefore suggesting achievable real-time performance by employing parallelism.

Accelerating quantum error correction algorithms through external computing devices such as FPGA has been a relatively new approach and provides many advantages with respect to an ASIC implementation. First of all, reconfigurability, thanks to a vast set of software tools that help the design and the optimization processes. This is unfortunately paid in terms of a reduction of performance due to the generic hardware of FPGAs, which are designed for the rearrangement of resources and the interconnections between them. An application-specific integrated circuit (ASIC) instead, is much more optimized in terms of delay and power consumption as well as the overall area occupation. On the other hand, ASICs require much higher costs and complex design/fabrication processes since they eventually require transistor-level optimization and appositely designed masks to be used in foundries, which are very expensive and worth only for large market scales. FPGAs instead, are ready-to-use devices, with a very fast integration in the overall architecture.

Many solutions have been proposed for real-time quantum error correction running on FPGA [4][12][5][11][18]. LILLIPUT [4], from Google Quantum AI, implements the Fowler [8][9] MWPM algorithm by computing offline all possible error events and programming the LUTs (Look-Up Table) with the error assignments. These LUTs are then accessed during the running mode of the decoder by using the received syndromes. This kind of approach has very high performance in terms of decoding latency without sacrificing accuracy, since the only cost is the accessing time of LUTs. On the other hand, the code distance is limited to $d = 5$, since as it increases, also the number of required LUTs would dramatically increase. Another approach is HELIOS [12] which instead uses a direct implementation of the full Union Find (UF) decoding algorithm on FPGA, without relying on an offline pre-computation. This kind of approach loses performance in terms of latency and accuracy compared to LILLIPUT's MPWM but gains in terms of scalability and implementable code distance, with values up to $d = 15$ and even larger with networks of FPGAs.

The goal of this work is to contribute to the implementation of a full MWPM decoding algorithm, in the Sparse Blossom [10] version, without relying on offline pre-computation, in order to obtain a highly-scalable and accurate decoder while fitting the real-time window requirements.

3 The Sparse Blossom Algorithm

The *Sparse Blossom* algorithm [10] is a version of the Minimum Weight Perfect Matching (MWPM) which graphically solves the decoding graph by assigning growing regions to each triggered detector node and, depending on the interactions (collisions) between these regions, extracts the edges and the related error patterns that may have caused the syndrome. This approach is highly scalable, with a distance $d = 29$ code being executed in $3.5\mu s$ on a single core of an Apple M1 Max processor, as reported in [10]. Yet, such latency still does not meet the time constraints required by quantum error correction.

3.1 Basic Concepts And Building Blocks

To better understand the Sparse Blossom algorithm, some basic concepts are presented:

- The first concept to be introduced is the *graph fill region* R, to which is assigned a certain radius y_R. This region has the goal of 'exploring' the detector graph and is linked to all the nodes and the edges within its source and its radius. Its source can be either a single detector event or other graph-fill regions that merge in a *blossom*. Moreover, it is assigned to each region a state: $+1$ (growing), -1 (shrinking), or 0 (frozen) which set the growth (or shrink) rate of the region radius. It must be noted that the velocity at which regions grow and shrink is constant and equal for every region.

- A *region edge* (R_a, R_b) describes the relationship between two graph fill regions R_a and R_b, or between a region and the boundary node. It includes its endpoints R_a and R_b as well as the shortest path between any detection event in R_a and any detection event in R_b.

- A pair of two regions R_a and R_b are matched to each other creating a *match*. The two regions must be touching, have a growth rate of zero (both frozen), and be linked by a region edge, also called *match edge*.

At the beginning of the algorithm, all regions are unmatched and growing. In the end, every region is matched, either to another region or the virtual boundary node. Some more complex building blocks are now presented:

- An *alterneting tree* is a tree where each node is associated with an active graph fill region and each edge, called *tree edge*, corresponds to a region edge. All the regions in the tree must touch each other and there is always at least one growing region inside the tree.

- A *blossom* is obtained when two growing regions belonging to the same alternating tree touch each other thus forming a region containing an

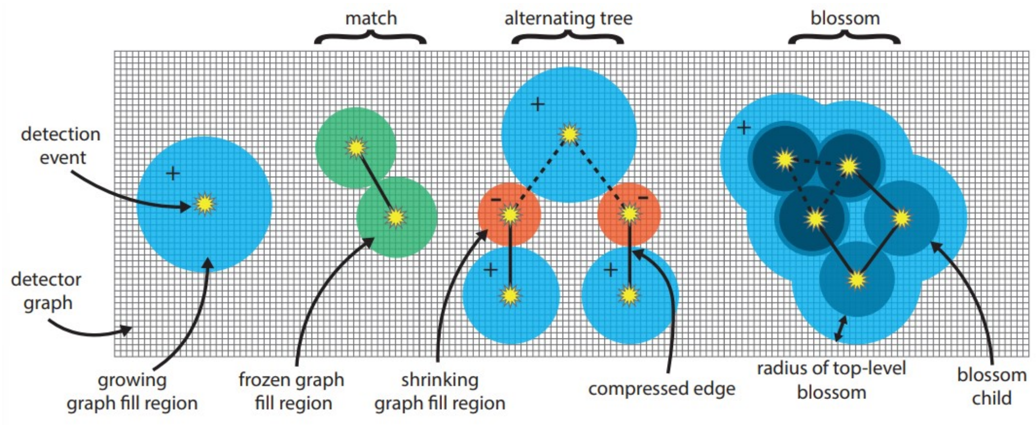


Figure 3: Fundamental structures in Sparse Blossom. [10]

odd-length cycle of regions. A region composing the blossom is called *blossom-children*.

A visual representation of all these structures is shown in Figure 3.

3.2 Algorithm Architecture

During the whole execution, the algorithm follows a *timeline*, with t always increasing, and processes the different events in the order they occur. The events that can occur are, for example, a region arriving at one node, colliding with another region, collapsing if it is a shrinking region reaching zero radius, and many more. During the execution, many building blocks and structures are created, modified, and deleted, depending on the events that occur. The algorithm goes on until there are no more events to be processed. Furthermore, the timeline of the events is based on the radius equation of the regions and structures involved, which computes the time at which the event will happen.

The algorithm is composed of 3 main blocks: a *matcher*, responsible for the management of alternating trees and blossoms structures; a *flooder*, which handles how regions evolve and the interaction events between them; and a *tracker*, which handles the timeline of all different events and makes sure that the other two blocks work with the correct order of events by using a single priority queue.

- The **matcher** is responsible for the management of alternating trees and blossoms structures, independently of how the detector graph is composed. At $t = 0$, every triggered detector node is the source of a growing region. By increasing their radius, the regions explore the graph, looking for other regions to collide and match with, including the virtual boundary node. A growing region can never hit a shrinking one since its radius changes with the same speed but in opposite directions. When the flooder notices that a collision happened between two growing regions, or a growing region and a frozen one, or the virtual boundary, a *collision edge* is found and it is given to the matcher which eventually changes the structure of an alternating tree or a blossom or simply add a new match. Once the algorithm has completed its execution the matcher provides all the pairs of matched detection events.

Visual Description. Intuitively, the algorithm is graphically applying what has been anticipated in the Decoding Graph paragraph 2.5: in fact, as regions expand from the initial sources (triggered detectors), they are exploring the graph for the nearest triggered detector region to pair with. When two regions hit each other they freeze (stops growing) and create a match. The edge between them will be the error that has caused the triggering of two detectors associated with the matched regions (also being the nodes at the extremes of the edge). Indeed, minimizing the path between two nodes corresponds to finding the most probable causing error.

To make another example, as it will be seen ahead when describing all the types of events that can occur (Figure 4), when a pair of two matched regions are hit by another expanding region, the pair is not matched anymore, and all three create a new alternating tree. This makes sense, since a group with an odd number of detector nodes implies ambiguity in the error choice, from the moment that the middle node would have to deal with two possible collision edges, whereas a physical error occurring can trigger just one pair of detectors. Therefore, the alternating tree is not frozen (as happens for matched regions) but continues to grow and explore the graph in order to solve the conflict.

- The **flooder** is responsible for managing how regions grow, shrink, or collide in the detector graph, regardless of structures such as alternating trees or blossoms (which instead are managed by the matcher). There are many *flooder events* that can happen:

  1. ARRIVE: a growing region can grow into an empty detector node.
  2. LEAVE: a shrinking region can leave a detector node.

3. COLLIDE: a growing region can hit another region or the boundary.
4. IMPLODE: a shrinking region can reach radius zero.

ARRIVE and LEAVE events do not change any structure and therefore are not notified to the matcher. On the contrary COLLIDE and IMPLODE change the structures and therefore are notified to the matcher which proceeds to update them as well as modify the growing rate of the regions. All the events are always notified to the tracker, which ensures that all of them are handled by the modules with the correct timing.

- The **tracker** ensures that flooder events are handled with the correct order by employing a *priority queue*. If some events change the structures and therefore the flooder predictions, the tracker eventually discards outdated events or asks the flooder to check if they are still valid. An example of a timeline is provided in Figure 5.

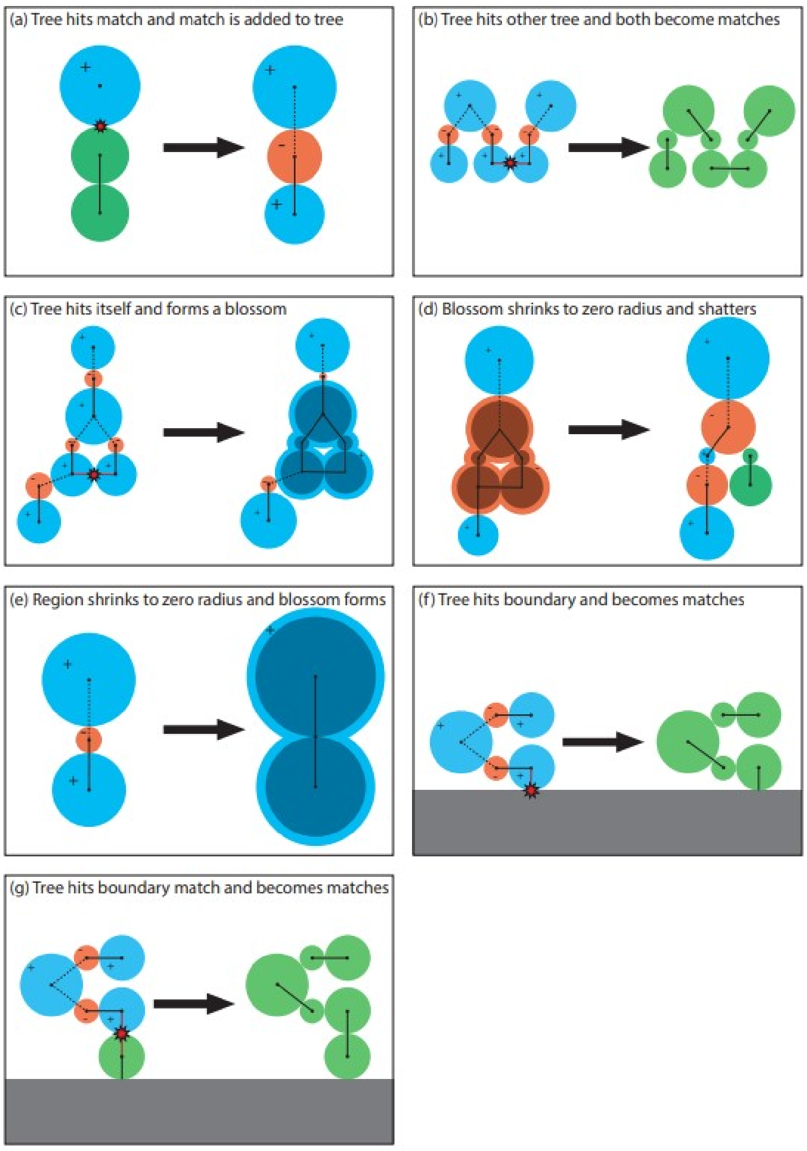
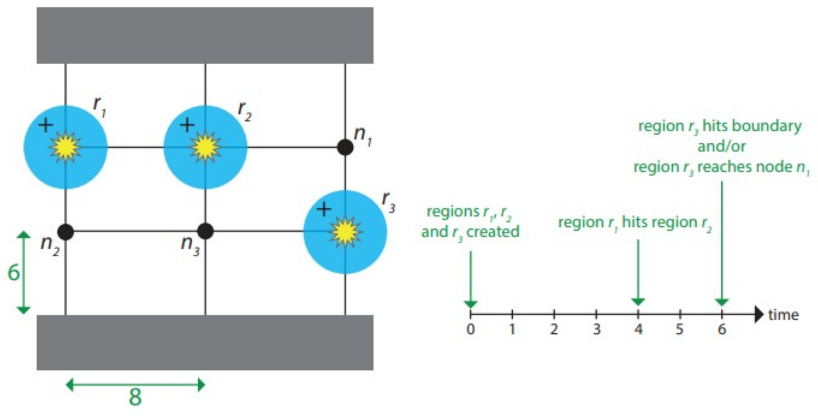

(a) Tree hits match and match is added to tree
(b) Tree hits other tree and both become matches
(c) Tree hits itself and forms a blossom
(d) Blossom shrinks to zero radius and shatters
(e) Region shrinks to zero radius and blossom forms
(f) Tree hits boundary and becomes matches
(g) Tree hits boundary match and becomes matches

Figure 4: Regions interaction events in Sparse Blossom. Background detectors have been omitted. Each node corresponds to a detection event. [10]



r_1
r_2
r_3
n_1
n_2
n_3
6
8
regions r_1, r_2 and r_3 created
region r_1 hits region r_2
region r_3 hits boundary and/or region r_3 reaches node n_1
time
0
1
2
3
4
5
6

Figure 5: Example of flooder events in timeline. [10]

4 Implementing Sparse Blossom with FPGAs

Real-time quantum error correction is fundamental to drastically reducing the error rates in quantum computers. The overall hardware architecture for QEC includes two counterparts: the quantum system and the classical computing device. The quantum system is realized with a 2D lattice of physical qubits, some of them being data qubits, which implement information redundancy; others parity qubits, also called detector qubits, which allow us to gain information about errors occurring without making the data qubits collapse due to the measurements. By reading all the detector qubits a syndrome is obtained. The syndrome is then forwarded to the classical computing device, which is composed of a decoding module and a correction module. The decoding module receives the syndrome and provides, depending on the implemented algorithm, the most probable set of errors that may have caused that syndrome. The correction module instead receives the error pattern extracted by the decoder and computes a correction to be provided back to the quantum system. The whole process is called *round of correction* and it must be performed within a time window of $1\mu s$ to allow a real-time implementation.

The proposed work focuses on contributing to achieving a hardware implementation of the Sparse Blossom decoding algorithm for real-time quantum error correction. As already discussed in Section 2.6, several FPGA implementations [4][12][5][11][18] of QEC algorithms have been realized. Among these, two main approaches have been pursued: *offline decoding* and *online decoding*.

- *Offline decoding* divides the entire decoding process into two parts.

First, an offline pre-computation of all possible error patterns that can be produced by a received syndrome. Then these patterns, which correspond to the outcome of the decoding algorithm itself, are stored in a local memory on the classical computing device, to be later accessed fastly during execution. This is the approach pursued by LILLIPUT [4] from Google Quantum AI which proposes a LUTs-based decoder to implement the Mimimum-Weight Perfect Matching. This kind of approach has very good performance in terms of latency since the only costs to be paid are the accessing time of LUTs. Moreover, it preserves the accuracy of the MWPM. On the other hand, this kind of approach explodes in terms of resources as the number of physical qubits and the distance of the code increase. As the number of physical qubits increases, also the decoding graphs become more and more populated of detector nodes, therefore increasing the complexity of the decoding problem. Indeed, LILLIPUT is able to achieve very good performances $\sim 40ns$ of latency for a $d = 5$ distance code.

- *Online decoding* instead, directly implements the whole algorithm on the classical computing device and therefore does not need any kind of pre-computation. Thy syndrome is received and the detector graph is extracted. Then all the steps of the chosen algorithm are executed by different units implemented in the device, and the best matching is obtained. This is the approach pursued by HELIOS [12], which implements on FPGA a Union-Find decoder. This kind of architecture loses performance in terms of latency since all the decoding steps must be performed on the board. Moreover, with Union-Find, also the accuracy is less with respect to an MWPM decoder, being the first derived from an approximation of the latter. However, the implementation gain in terms of scalability and becomes able to implement surface codes with a much larger number of physical qubits, as the architecture of the decoder is not limited anymore by the memory resources. As it is claimed in [12], HELIOS decoder implements surface codes with distance $d = 7$, with a claimed average latency in the order of $\sim 120ns$, and potentially go up to $d = 15$ by employing more FPGAs with higher performance.

4.1 Accelerating The Sparse Blossom

We aim to contribute to the implementation of an online Sparse Blossom decoder, fully integrated on FPGA which does not rely on any kind of pre-computation of the input syndromes, in order to obtain a highly-scalable decoder, able to perform real-time decoding on large distance surface codes.

The proposed work, **QuErK** (**Qu**antum **Er**ror Correction **K**ernel), is based on the open-source code *PyMatching 2* presented in [10], which implements the Sparse Blossom version of the Fowler[9][8] MWPM decoding algorithm. The latest version of PyMatching performs the decoding of $d = 29$ surface code in approximately $3.5\mu s$ on M1 Max Processor, while keeping the accuracy of an MWPM algorithm.

The published open-source code is very complex, with a lot of functions and dynamic data structures which are not suitable for high-level synthesis. In Figure 6, it is reported the tree of all functions called during the execution of the *C++* code:

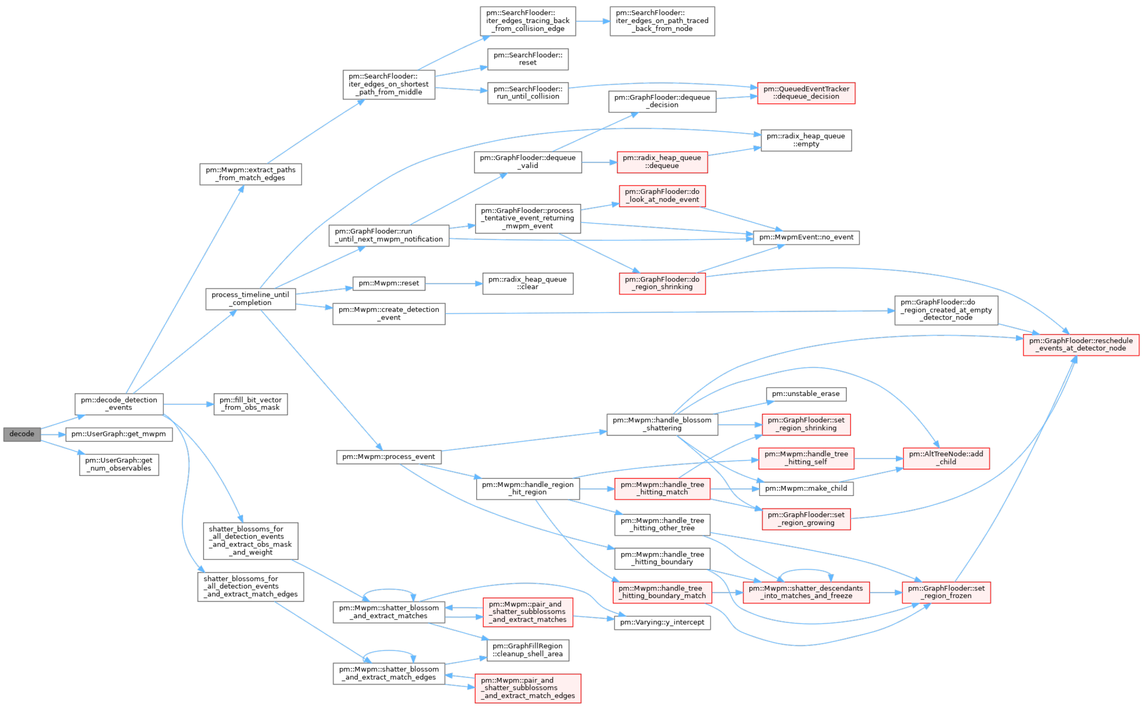

decode
pm::decode_detection _events
pm::UserGraph::get_mwpm
pm::UserGraph::get _num_observables
pm::fill_bit_vector _from_obs_mask
pm::Mwpm::extract_paths _from_match_edges
process_timeline_until _completion
shatter_blossoms_for _all_detection_events _and_extract_obs_mask _and_weight
shatter_blossoms_for _all_detection_events _and_extract_match_edges
pm::SearchFlooder:: iter_edges_on_shortest _path_from_middle
pm::SearchFlooder:: iter_edges_tracing_back _from_collision_edge
pm::SearchFlooder:: reset
pm::SearchFlooder:: run_until_collision
pm::SearchFlooder:: iter_edges_on_path_traced _back_from_node
pm::GraphFlooder::dequeue _decision
pm::QueuedEventTracker ::dequeue_decision
pm::GraphFlooder::dequeue _valid
pm::radix_heap_queue ::dequeue
pm::radix_heap_queue ::empty
pm::GraphFlooder::run _until_next_mwpm_notification
pm::GraphFlooder::process _tentative_event_returning _mwpm_event
pm::GraphFlooder::do _look_at_node_event
pm::MwpmEvent::no_event
pm::Mwpm::reset
pm::radix_heap_queue ::clear
pm::GraphFlooder::do _region_shrinking
pm::Mwpm::create_detection _event
pm::GraphFlooder::do _region_created_at_empty _detector_node
pm::GraphFlooder::reschedule _events_at_detector_node
pm::unstable_erase
pm::Mwpm::handle_blossom _shattering
pm::GraphFlooder::set _region_shrinking
pm::Mwpm::process_event
pm::Mwpm::handle_tree _hitting_self
pm::AltTreeNode::add _child
pm::Mwpm::handle_region _hit_region
pm::Mwpm::handle_tree _hitting_match
pm::Mwpm::make_child
pm::GraphFlooder::set _region_growing
pm::Mwpm::handle_tree _hitting_other_tree
pm::Mwpm::handle_tree _hitting_boundary
pm::Mwpm::handle_tree _hitting_boundary_match
pm::Mwpm::shatter_descendants _into_matches_and_freeze
pm::GraphFlooder::set _region_frozen
pm::Mwpm::shatter_blossom _and_extract_matches
pm::Mwpm::pair_and _shatter_subblossoms _and_extract_matches
pm::Varying::y_intercept
pm::GraphFillRegion ::cleanup_shell_area
pm::Mwpm::shatter_blossom _and_extract_match_edges
pm::Mwpm::pair_and _shatter_subblossoms _and_extract_match_edges

Figure 6: Functions tree of the Sparse Blossom.

The tree is full of dynamic memory data structures, recursive calls, and pointers to pointers. Therefore a straightforward implementation of the whole algorithm appears to be full of difficulties and requires to basically re-think and re-write the entire structure0 from scratch. Consequently, by performing an inspection of the *pseudo-code* and by employing a timing analysis of the running algorithm with *Valgrind*[19], we proceeded to identify bottlenecks and the most time-consuming functions in the tree. By looking at the graph in Figure 6 on the far right it can be seen a function, belonging to the *graph flooder* module, which is responsible for the computation of regions expansions and time of collision between them. It is the end of many branches in the tree and therefore is expected to have a large number of calls. By checking the *Valgrind* Analysis for *percentage time occupation* of the functions it has been confirmed that the function occupies the 30.64% of the whole execution time, as it is also shown in Figure 7.

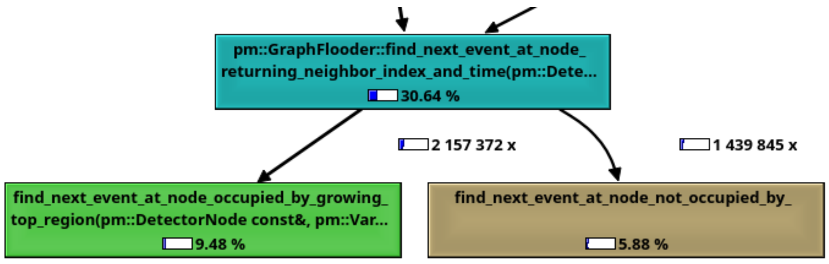

pm::GraphFlooder::find_next_event_at_node_returning_neighbor_index_and_time(pm::Dete...
30.64 %
2 157 372 x
1 439 845 x
find_next_event_at_node_occupied_by_growing_top_region(pm::DetectorNode const&, pm::Var...
9.48 %
find_next_event_at_node_not_occupied_by_
5.88 %

Figure 7: Percentage time occupation. [19]

4.2 The Flooder find_next_event Component

The function *find_next_event* belongs to the *Graph Flooder* module and it is responsible for the computation of the next collision event between two regions. It receives as input one *detector node* and all its *neighbours*. Then, it computes the interactions and the collision times between the received node and all the possible neighbors, including the virtual boundary node, and returns as output the event happening first, which means the event with the lowest collision time. The function provides to the output also the index of the neighbor participating in that event, as well as the collision time itself; fundamental information to be later forwarded to the matcher and the tracker, for alternating tree structure modification and queue timeline re-ordering respectively.

The original version of the function proceeds in the following steps, for each detector node and neighbor node pair:

1. Two triggered detector nodes are read from the memory.

2. The radius of both nodes is computed.

3. Evaluation of collision condition (whether or not a collision will happen). For example, a collision will happen if at least one of the regions associated with the nodes is growing and the other is not shrinking. If one region is growing and the other is shrinking, as already mentioned,

the radius expansion and contraction happen with the same, but opposite, rate, and therefore there is no collision. If is that the case, the following steps are not executed, and a new pair of nodes is processed.

4. Computation of the basic collision time (computed considering only the first node growing.

5. Evaluation of condition that checks if also the neighbor region is growing.

6. If both regions are growing, then the collision time is divided by two since the two regions are expanding toward each other.

7. At the first iteration the computed collision time, and the index of the first neighbor are stored. In the following iterations, these values are updated exclusively if the new collision time is lower than the stored one. Basically, it is saved and eventually updated, the minimum collision time and the index of the neighbor belonging to that collision event.

The impact of this function becomes more and more time-consuming as the number of nodes increases in the detector graph since every node and its neighbors must be explored.

In Algorithm 1, we present the pseudocode describing such a component.

```
input  : An expanding or shrinking detector node: x
output: Collision Time, Collided Node
collisionTime ← Inf;
collidedNode ← nil;
foreach neighboring node: y do
    radx ← ComputeRadius($x$);
    rady ← ComputeRadius($y$);
    if IsShrinking($x$) or IsShrinking ($y$) then
        #collision condition;
        continue;
    end
    if IsGrowing($x$) then
        #collision time;
        currentTime ← ComputeTime (x,y);
        if IsGrowing ($y$) then
        currentTime ←currentTime /2;
    end
    if currentTime < collisionTime then
        #collision time update;
        collisionTime ← currentTime;
        collidedNode ← y;
    end
end
```
Algorithm 1: Procedure describing *find_next_event* component

4.3 Implementation As A Dataflow Architecture

We decided to implement a *dataflow* architecture to accelerate the aforementioned function *find_next_event*, by analyzing the possibilities of parallelism and finding the independent tasks in the processing chain.

The inputs of the accelerator correspond to the required input of the Flooder component:

1. detector node: corresponds to the index of the nodes to consider in the decoding graph;
2. num_nodes: number of nodes in the graph;
3. num_regions: number of regions in the graph;
4. radius: vector containing the radius of all the regions considered;

5. region_that_arrived_top: parent region for any given node in the graph;
6. wrapped_radius_cached: precomputed radius for a variety of useful computation and for the maximization of the performance;
7. neighbors: matrix defining, for each detector node, the corresponding neighbors in the graph;
8. neighbor_weights: matrix defining, for each neighbor relationship, the corresponding weight;
9. neighbor_observables: matrix defining, for each neighbor relationship, the corresponding logical observables;
10. out_neighbor: neighbor index to give as output;
11. out_time: time to give as output.

Indeed, we proceeded to achieve a task-level parallelism, by allowing the parallel execution of the radius computation step and collision condition evaluation step, which are independent of each other and can be executed simultaneously. Moreover, also the basic collision time computation and its update condition have been reorganized to be executed together. The scheme of the dataflow architecture is reported in Figure 8.

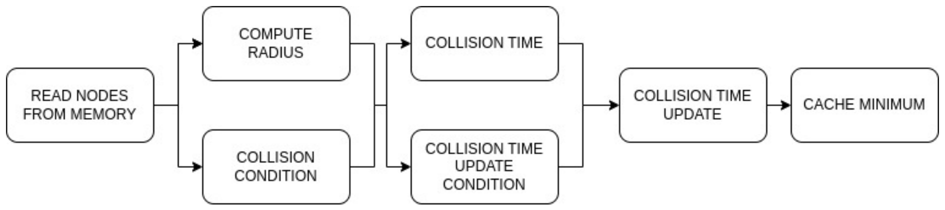


Figure 8: Dataflow architecture.

Each component maps to one of the tasks described in Section 4.2, while exploiting the absence of specific data dependencies to compute specific jobs in parallel. Indeed:

- the components *COMPUTE RADIUS* and *COLLISION CONDITION* are independent, but both require data obtained at the *READ NODES FROM MEMORY* stage;
- the *COLLISION TIME* and *COLLISION TIME UPDATE CONDITION* both depend on *COMPUTE RADIUS*, while the second one also depends on *COLLISION CONDITION*;

- *COLLISION TIME UPDATE* and *CACHE MINIMUM* need to be executed one after the other, in a sequential fashion, due to their data dependencies.

.

At each time step, the dataflow receives and consumes one of the neighbors of the node, and updates, in the *CACHE MINIMUM* stage, the local minimum found. Once all the neighbors are consumed, the final minimum is global and equivalent to the correct output of the component.

In order to achieve data-level parallelism, our proposed implementation also includes pipelining, to allow the single stages of the processing chain to be fully operational and always working. Indeed, the evaluations of the neighbors are all independent one with respect to the others.

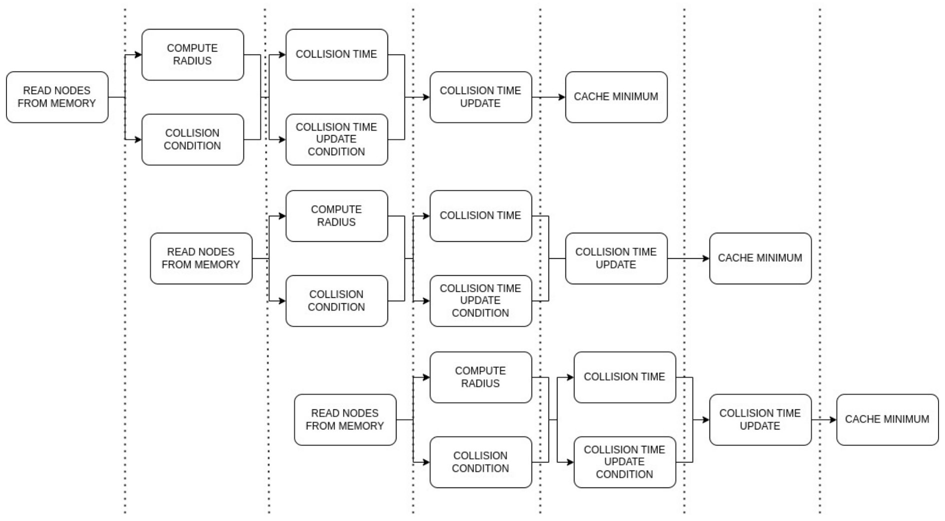


Figure 9: Pipeline execution.

Finally, in order to reduce the amount of time required for memory transfer, we supported High-Bandwidth Memory (HBM), as a tool to reduce the overall latency.

5 Results

In this section, we describe the experimental setup and results we obtained for the testing of the architecture, thus validating our solution and showing its advantages and disadvantages. In particular, we show that it is about $50x$ faster than software solution, and $118x$ more energy efficient.

5.1 Experimental Setup

Our setup consisted of a system with *AMD Ryzen 7 3700X* and 32 *GB of RAM*. We used the AMD-Xilinx Alveo U55C High Performance Compute Card [21], shown in Figure 10, with 1,304K Look-Up Tables (LUTs), 2,607K registers, 9,024 DSP slices, and 16 GB of HBM Memory.

Figure 10: Testing setup: host code on *AMD Ryzen 7 3700X* and 32 *GB of RAM* plus acceleration on *AMD-Xilinx Alveo U55C* [21]

This device has been chosen due to its adherence to the requirements for quantum error correction: it allows for high performance, a noticeable amount of available resources, and support for HBM.

For the synthesis, we used Vitis HLS and Vitis version 2022.2.

5.2 Required Hardware Resources

QuErK uses mostly BRAMs (897, about 50% of the total), since it requires storing information related to the neighbors of the given node. The usage of HBM allows for fast memory transfer, while the algorithm requires a reduced amount of LUTs and DSPs, to the simplicity of the operations internal to the stages, which are highly repetitive and redundant, yet applied to a noticeable amount of data. The overall frequency used for synthesis is 300 MHz. In Table 2, we present the amount and the percentage of used resources for *AMD-Xilinx Alveo U55C*.

Table 2: Resources usage

Frequency	**LUT**	**LUTAsMem**	**REG**	**BRAM**	**DSP**
300 MHz	14,354 (1,22 %)	2,377 (0,40 %)	16,109 (0,66%)	897 (49,34%)	0 (0,00%)

5.3 Execution time

To evaluate execution time, a host written in OpenCL produces synthetic data that corresponds to a given decoding graph, with a given amount of nodes.

In order to compare the performance achieved by the hardware accelerator with respect to the non-accelerated software, we run the same experiments both on a pure software component in Python and on the host, by using the same CPU architecture and available RAM memory.

To measure execution times, we tested detectors graphs whose size varies in the range of $20‘000 - 110‘000$ nodes, as shown in Figure 11, thus corresponding to average-case quantum computers. The retrieved amount of time for the hardware acceleration ranges between $0.0002 - 0.0018s$, due to the high frequency of data transfers between the host code and the accelerated function, the dataflow architecture, and pipelining, that exploited at most the data-level and task-level parallelism.

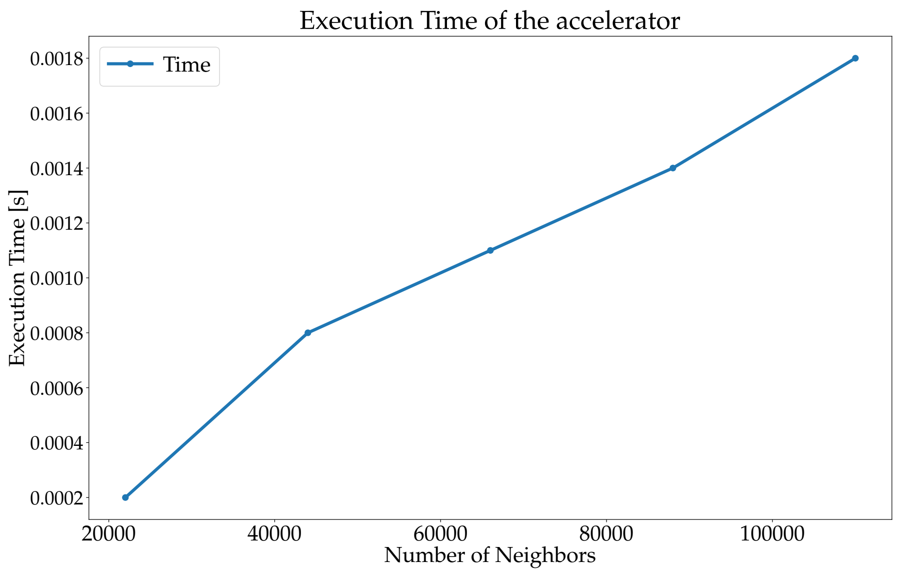


Figure 11: Execution time for the hardware accelerator on *AMD-Xilinx Alveo U55C* [21]

On the other hand, the execution time required by the pure software varies noticeably according to the size of the problem, thus letting the accelerator reach a $49x$ speedup with respect to the software implementation.

5.4 Energy Efficiency

We also validated the energy efficiency of our solution, compared to the pure software version. To do so, we established the following metric to evaluate energy efficiency:

$$EE = \frac{Throughput}{PowerConsumption} = \frac{numberofneighbors}{ExecutionTime \cdot PowerConsumption}$$

We evaluated such a metric for both the hardware accelerator and the software version, and show the results in Figure 12, by using a logarithmic scale. As can be noticed, the energy efficiency of the hardware accelerator is higher than the software version by orders of magnitude. Indeed, both the execution time and the power consumption are lower. The values for the power consumption are generally close to $21,8W$, while the power consumption of the CPU *AMD Ryzen 7 3700X* is close to $65W$. Therefore, our solution provides an accelerator that is, at the same time, faster than software, and more energy efficient, thus approaching the requirements given by a quantum hardware management system, that requires reduced power dissipation to keep necessary low temperatures. We also notice that the energy efficiency is up to $118x$ higher for the hardware accelerator with respect to the software version.

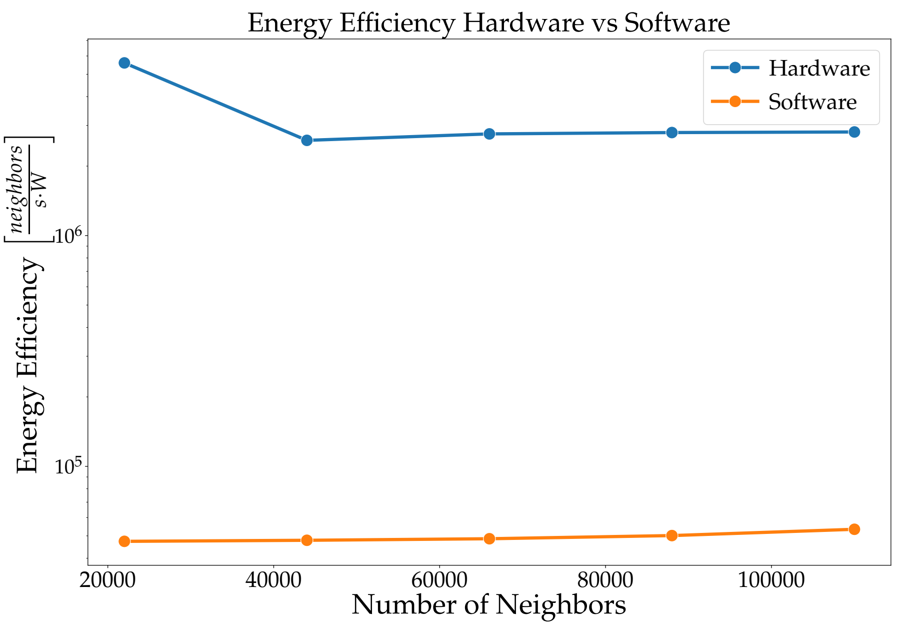


Figure 12: Energy Efficiency of the hardware accelerator with respect to the software version, by using board *AMD-Xilinx Alveo U55C* [21]

6 Conclusions And Future Developments

The presented report shows the huge potential and the reasoning behind Quantum Computing, as well as an introduction to the basic concepts of Quantum Error Correction and the strategies employed for the realization of QEC architectures with real-time performance. Surface codes are largely employed in error correction to encode logical qubits with many more physical qubits and drastically lower the error rates. They require a decoding algorithm to extract from the syndromes the most probable errors patterns in order to compute a correction to be provided back to the Quantum Computer. Among these algorithms, the Sparse Blossom is very promising, since it is able to keep the accuracy of a MWPM while still meeting real-time performance constraints. It is then proposed an architecture for the acceleration of a graph flooder function *find_next_event*, with a system capable of receiving a syndrome and processing it with the host code. The host code then communicates with the accelerated block every time the *find_next_event* function is called. This is the first step towards the full integration of the algorithm on board, which will drastically reduce execution time, a fundamental requirement to meet the $1\mu s$ constraint per round of correction, needed to perform real-time error correction Additionally, the very promising speedup can be further pushed by implementing multiple pipelines and using all the BRAM resources in the system. The complete integration of the algorithm requires to completely rethink the main blocks such as FLOODER, TRACKER, and MATCHER, as well as all the interconnections between them; but it is the right approach to achieve real-time Quantum Error Correction.

References

[1] Google Quantum AI. *Quantum computer datasheet.* `https://quantumai.google/hardware/datasheet/weber.pdf`. [Online; accessed 21-June-2023]. 2021.

[2] Beatrice Branchini et al. "A Bird's Eye View on Quantum Computing: Current and Future Trends". In: *IEEE EUROCON 2023 International Conference on Smart Technologies.* 2023, pp. 1–7.

[3] Sergey Bravyi, Martin Suchara, and Alexander Vargo. "Efficient algorithms for maximum likelihood decoding in the surface code". In: *Physical Review A* 90.3 (Sept. 2014). DOI: `10.1103/physreva.90.032326`. URL: `https://doi.org/10.1103%2Fphysreva.90.032326`.

[4] Poulami Das, Aditya Locharla, and Cody Jones. *LILLIPUT: A Lightweight Low-Latency Lookup-Table Based Decoder for Near-term Quantum Error Correction.* 2021. arXiv: `2108.06569 [quant-ph]`.

[5] Poulami Das et al. *A Scalable Decoder Micro-architecture for Fault-Tolerant Quantum Computing.* 2020. arXiv: `2001.06598 [quant-ph]`.

[6] Nicolas Delfosse and Naomi H. Nickerson. "Almost-linear time decoding algorithm for topological codes". In: *Quantum* 5 (Dec. 2021), p. 595. DOI: `10.22331/q-2021-12-02-595`. URL: `https://doi.org/10.22331%2Fq-2021-12-02-595`.

[7] Marc Duranton et al. *HiPEAC Vision 2023.* Jan. 2023, pp. 1–238. URL: `https://inria.hal.science/hal-04023794`.

[8] Austin G. Fowler. *Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time.* 2014. arXiv: `1307.1740 [quant-ph]`.

[9] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86.3 (Sept. 2012). DOI: `10.1103/physreva.86.032324`. URL: `https://doi.org/10.1103%2Fphysreva.86.032324`.

[10] Oscar Higgott and Craig Gidney. *Sparse Blossom: correcting a million errors per core second with minimum-weight matching.* 2023. arXiv: 2303.15933 [quant-ph].

[11] Adam Holmes et al. *NISQ+: Boosting quantum computing power by approximating quantum error correction.* 2020. arXiv: 2004.04794 [quant-ph].

[12] Namitha Liyanage et al. *Scalable Quantum Error Correction for Surface Codes using FPGA.* 2023. arXiv: 2301.08419 [quant-ph].

[13] Kai Meinerz, Chae-Yeun Park, and Simon Trebst. "Scalable Neural Decoder for Topological Surface Codes". In: *Physical Review Letters* 128.8 (Feb. 2022). DOI: 10.1103/physrevlett.128.080505. URL: https://doi.org/10.1103%2Fphysrevlett.128.080505.

[14] Ramon Overwater. *Data for: Neural Network Decoders for Surface Codes.* 2022. DOI: 10.4121/16539786.v1. URL: https://data.4tu.nl/articles/dataset/Data_for_Neural_Network_Decoders_for_Surface_Codes/16539786/1.

[15] Varsamopoulos S. *Neural Network based Decoders for the Surface Code.* 2019. DOI: 10.4233/uuid:dc73e1ff-0496-459a-986f-de37f7f250c9. URL: https://doi.org/10.4233/uuid:dc73e1ff-0496-459a-986f-de37f7f250c9.

[16] John Shalf. "The future of computing beyond Moore's Law". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378 (Jan. 2020), p. 20190061. DOI: 10.1098/rsta.2019.0061.

[17] Neereja Sundaresan et al. *Matching and maximum likelihood decoding of a multi-round subsystem quantum error correction experiment.* 2022. arXiv: 2203.07205 [quant-ph].

[18] Yosuke Ueno et al. "QECOOL: On-Line Quantum Error Correction with a Superconducting Decoder for Surface Code". In: *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE, Dec. 2021. DOI: 10.1109/dac18074.2021.9586326. URL: https://doi.org/10.1109%5C%2Fdac18074.2021.9586326.

[19] *Valgrind.* https://valgrind.org/. [Online; accessed 21-June-2023].

[20] Yue Wu, Namitha Liyanage, and Lin Zhong. *An interpretation of Union-Find Decoder on Weighted Graphs.* 2022. arXiv: 2211.03288 [quant-ph].

[21] *Xilinx Alveo U55C.* https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html. [Online; accessed 21-June-2023].