# Quantum Error Correction on FPGAs

Filippo Corna, Davide Salonico

June 30, 2024

filippo.corna@mail.polimi.it davide.salonico@mail.polimi.it

**Abstract**

Quantum Computing is a new paradigm of computation that allows for an exponential speedup with respect to classical computing, over a noticeable number of scenarios. Unfortunately, current quantum hardware is affected by noise, which reduces performance. In this work, we aim to accelerate one of the state-of-the-art techniques for error correction: the Minimum-Weight Perfect Matching. We target FPGAs as a heterogeneous architecture that exploits task-level and data-level parallelism to reduce execution time.

## 1 Introduction

Quantum computing represents a new computing model in comparison to classical computing that, exploiting properties of quantum mechanics, significantly reduces time complexity to solve some specific class of problems. The implications of this technology can drastically revolutionize various fields of computing and beyond. However, there are many challenges to overcome in order to make quantum computers actually useful for complex tasks. Noise can significantly affect quantum computers due to the delicate nature of qubits and their reliance on quantum states. Qubits are highly sensitive to environmental disturbances, such as temperature fluctuations, electromagnetic radiation, and even cosmic rays. These disturbances can cause decoherence, where qubits lose their quantum properties like superposition and entanglement, leading to errors in calculations. Therefore, implementing effective Quantum Error Correction (QEC) is a critical challenge in advancing quantum computing technology. As well as classical error correction, QEC relies on the basic idea of encoding information in a redundant way, making possible, through a decoding algorithm, not only to detect errors presence, but also the corrections to be applied in order to continue the computation correctly.

In this work, we aim to implement and accelerate the state-of-the-art Sparse Blossom Algorithm (SBA) [4] on FPGAs for two main reasons: meeting the strict time constraint that the problem imposes and use a power efficient technology to avoid adding noise in the environment.

The paper is organized as follows: Section 2 provides background knowledge about QEC and a decoder algorithm called Sparse Blossom Algorithm, whereas in Section 3 is a code analysis is presented. In Section experimental setup and results are discussed. We conclude in Section 5 with some possible future developments for this work.

# 2 Background Knowledge

## 2.1 QEC Overview

One effective approach to quantum error correction involves the use surface codes arranging qubits on a two-dimensional lattice. In this setup, *data qubits* hold the actual quantum information, while *ancilla qubits* are used to detect and correct errors. Ancilla qubits are strategically placed around data qubits and interact with them through quantum gates to check for errors. These ancilla qubits are then measured to reveal any errors' presence, nature, and location: the result of the measurement is called *syndrome*. Using this information, a correction algorithm , the *decoder*, determines the necessary operations to rectify the errors on the data qubits without directly measuring them, thus preserving their quantum state. This cycle of error detection and correction is continuously repeated throughout the computation to ensure the quantum information remains accurate. Hence, using surface codes with data and ancilla bits is a promising method for achieving scalable and fault-tolerant quantum computing. Some families of error decoding algorithms exploit *decoding graphs* in which vertices represent parity qubits (also called *detectors*) and edges are related to the errors that trigger the adjacent detectors. A weight is assigned to each edge, representing the probability for the corresponding error to happen.

## 2.2 The Sparse Blossom Algorithm

The Sparse Blossom Algorithm is a fast implementation of the Minimum-Weight Perfect Matching (MWPM) decoder, the most widely used decoder for dealing with graph-like error detection models. It can be used with several important classes of quantum error correction codes, including surface codes[3]. This algorithm builds upon Jack Edmonds' original Blossom Algorithm [2], enhancing its efficiency specifically for sparse graph structures.
Below we list some of the main components of the SBA in order to better understand its key concepts. *Nodes* are the vertices of the graph, representing individual parity qubits that can be connected through edges. Each node contains essential information for the algorithm, such as the node's state, its containing region, and its connections with other nodes. When, in a syndrome, the value corresponding to a detection node is set to 1, that particular node is called *detection event*. *Graph fill regions* 1 represent exploratory regions around a detection event or the surface of other graph fill regions (blossoms). *Alternating trees* 2 are composed of active graph fill regions connected by region edges, containing at least one growing region and always

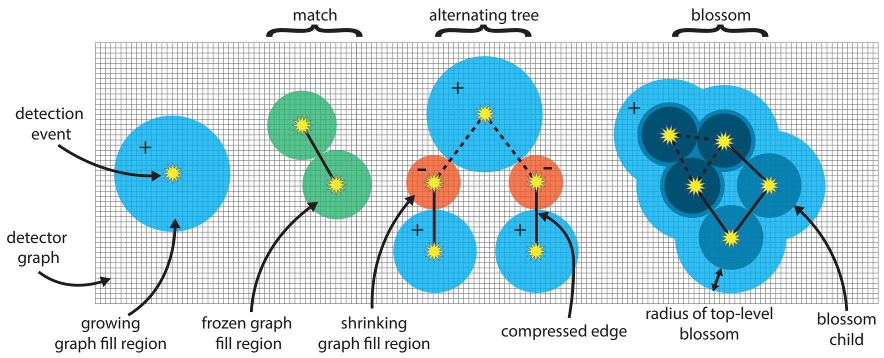


Figure 1: A graphic representation of SBA concepts

having one more growing region than shrinking regions. *Blossoms* form when two growing regions within the same alternating tree collide. They consist of an odd-length cycle of regions connected by region edges and can be nested, with blossoms having their own blossom-children.
These regions are analogous to nodes or blossoms in the standard blossom algorithm and their radii can grow, shrink, or stay constant. *Compressed edges* represent paths in the detector graph between detection events or between a detection event and the boundary. These edges contain all information relevant for error correction and they are stored in a compact form. Region edges describe relationships between graph fill regions or between a region and the boundary, comprised of compressed edges that represent shortest paths between detection events in the regions. *Matches* are pairs of touching regions with a zero growth rate, connected by a match edge. Initially, all regions are unmatched; by the algorithm's end, every region is matched.

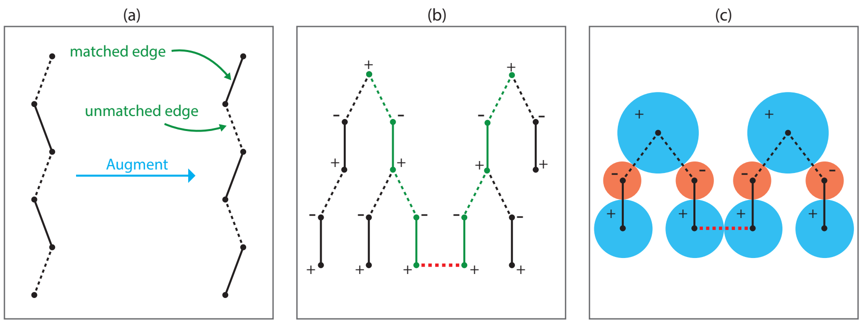


Figure 2: Representation of a) Augmenting path b) Alternating trees c) Alternating trees within regions

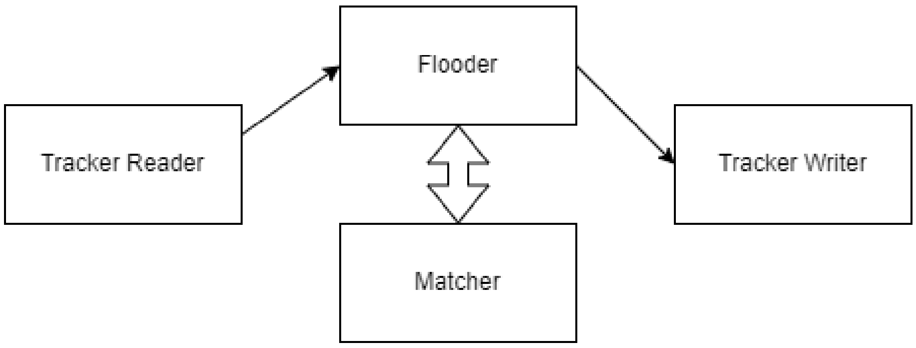


Figure 3: A high-level view of the principal components of SBA

This structured approach allows the sparse blossom algorithm to efficiently manage the complex relationships and interactions in the detector graph, optimizing performance for quantum error correction tasks. The current implementation of the SBA involves three key components: the *matcher*, the *flooder*, and the *tracker*3. The **flooder** is responsible for managing how graph fill regions grow, shrink or collide in the decoder graph, and is not concerned with the structure of the alternating trees and blossoms. Whenever two regions interact this information is passed to the **matcher**: the entity responsible for managing regions and blossoms interactions. It operates on a higher-level data structure and it can instruct the flooder to perform specific operations. The results of flooder computations are *events* that are stored in a priority queue managed by the **tracker** component. The final outcome is a Minimum-Weigh Perfect Matching from which corrections codes are straightforward to compute.

## 3 Code analysis and Design Process

Our starting point was the work described in "Towards the Acceleration of the Sparse Blossom Algorithm for Quantum Error Correction"[6] which focused on accelerating a subroutine which represented the bottleneck of the SBA. Our main goal is to pursue the acceleration to a wider number of components.
To study the SBA algorithm we created a detailed block diagram, in which each block represent a functional unit of the algorithm. The functional units must be thought as building-blocks of the hardware component that has been created on the FPGA. In the diagram blocks that required to retrieve data from main memory have been highlighted 4, while, when possible, the

data flow among the various components has been optimized for the sake of complexity optimization and speed. The state-of-the-art pure software implementation of the algorithm, named PyMatching, relies on many dynamically allocated data structures and recursive functions, which are not reproducible on a FPGA. To overcome the aformentioned problems we came up with two main optimization strategies: transforming all the recursive data structures in the software into their iterative counterpart (sometimes establishing an upperbound led by heuristcs to meet resource constraints of the board) and using an HLS level cache to exploit data locality in the algorithm.

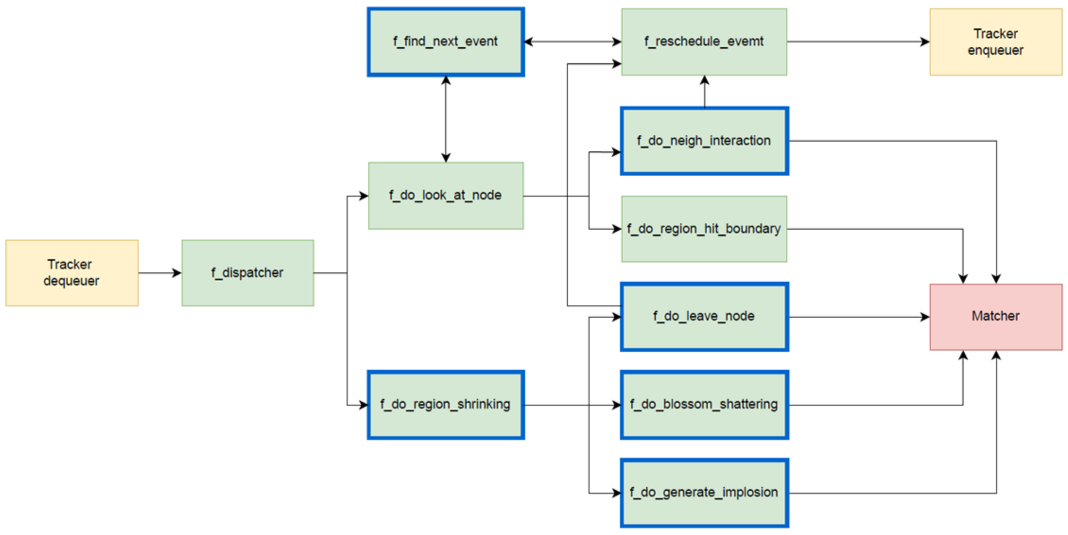


Figure 4: Flooder block-diagram analysis. Components highlighted in blue require access to memory

## 3.1 Flooder Analysis

We focused our efforts particularly on the Flooder. With a high level view, we can consider the flooder as a component that receives a *flood event* from the tracker and, after various computations and interfacing with other components, returns, if necessary, a *MWPM (Minimum Weight Perfect Matching) event* to the matcher.
Analyzing the code in more detailed manner, we noticed that the tracker is responsible for ensuring *flooder events* occur in the correct order. Once the

flooder starts handling these events, it first checks the type of event (specifically distinguishing between events related to nodes and events related to regions) and then processes the event. Using the same terminology as in the Sparse Blossom paper [4], if the flooder identifies a region growing into an empty node (ARRIVE) or a region shrinking and leaving a node (LEAVE), the flooder will ensure to insert these events into the tracker's priority queue. Otherwise, if the flooder identifies a region colliding with another region or a boundary (COLLIDE) or a region imploding (IMPLODE), the flooder will create an *MWPM event*, which it will be passed to the matcher, with all the necessary information to handle these situations.

## 3.2 DaCH

We chose to use the DaCH[1], a dataflow cache for high-level synthesis, in order to move data into On-Chip memory of the FPGA with the goal to reduce latency with respect to a Off-Chip memory solution5. Our choice is mainly motivated by the fact that this implementation fits well within the Xilinx guidelines for creating efficient HLS designs. In DaCH, each DRAM-mapped array can be associated with a level 2 (L2) cache with one or more ports, and each port can optionally provide a level 1 cache. The L2 cache runs in a separate dataflow task with respect to the application accessing it. In the acceleration of the SBA we set L1 cache parameters so that, according to specifications, the architecture will be a pure *Dataflow cache* and it will have smaller latency and less used resources on the reconfigurable fabric of the board. The L2 cache consumes a stream of requests (read or write) and produces a stream of responses (read), using a Load, Compute, Store (LCS) coding pattern. It interacts with a computing function in *master-slave* fashion: the cache acts as the slave while the computing function is the master and it actively produces the requests to be consumed.

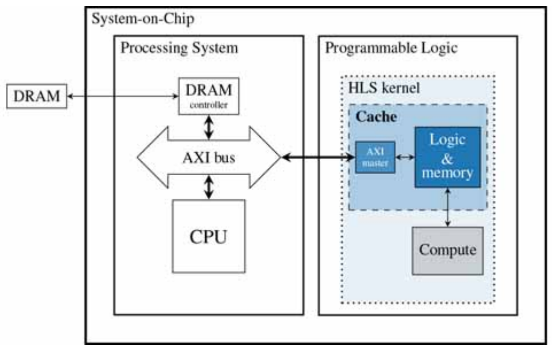
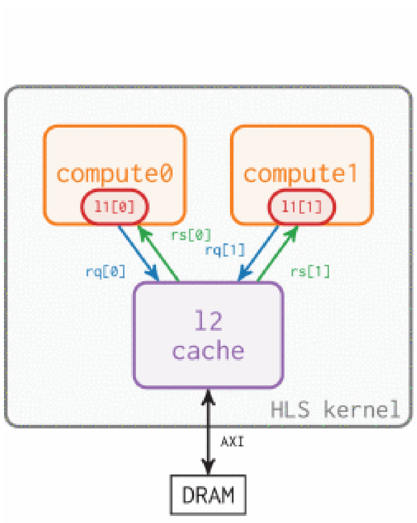
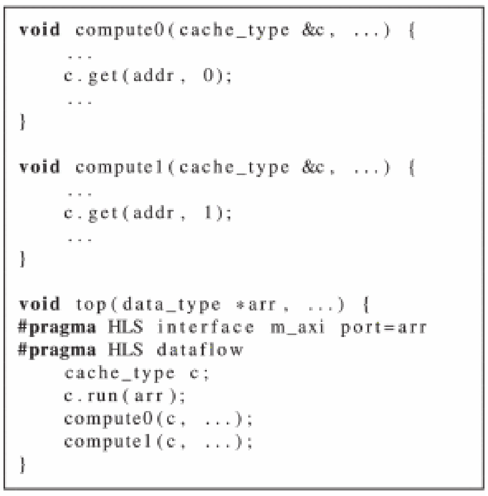


Figure 5: DaCH cache





```
void compute0(cache_type &c, ...) {
    ...
    c.get(addr, 0);
    ...
}

void compute1(cache_type &c, ...) {
    ...
    c.get(addr, 1);
    ...
}

void top(data_type *arr, ...) {
#pragma HLS interface m_axi port=arr
#pragma HLS dataflow
    cache_type c;
    c.run(arr);
    compute0(c, ...);
    compute1(c, ...);
}
```

(a) Dataflow graph. (b) Source code.

Figure 6: An high -level view of the interaction between L1 cache, L2 cache and two computing functions

Another useful characteristic of DaCH cache is that is fully customizable and it's easy to tune parameters in order to balance performance and resource usage. In addition, DaCH exposes a set of profiling functions that makes possible to measure performance indexes such as hit rate, miss rate, latency and others, both for L1 and L2 caches. Exploiting this profiling functions designers who include DaCH in their design simply need to perform a design space exploration of the cache configurations. Our work is focused on the usage of DaCH library because caching could be the only feasible memory optimization for algorithms with data-dependent or irregular memory access patterns, but with good data locality as for the SBA.

However, DaCH suffers of some issues during synthesis phase, especially when the entries of the cache have a certain size, even though it's not clear which is the threshold. In fact this cache implementation can be used only for a subset of data types in the algorithm.

## 4 Experimental Results

### 4.1 Experimental Setup

We implemented the design in HLS using AMD Xilinx Vitis HLS 2022.2, and used AMD Xilinx Vitis 2022.2 to generate the bitstream. Our target board is an AMD Xilinx Alveo U55C, with 16 GB of HBM, available both through the AMD Heterogeneous Accelerated Compute Cluster (HACC) program and DEIB - Dipartimento di Elettronica, Informazione e Bioingegneria's cluster at Politecnico di Milano.

### 4.2 Correctness

At the current state of the development only a fraction of the whole algorithm has been accelerated (the flooder). Hence, extensive testing is practically unfeasible due to the strong data dependence among data in the flooder and tracker's events. We manually tested a graph with reduced dimensions by modifying PyMatching source code in order to provide a detailed log of the events happening during the decoding phase. By hardcoding calls to the tracker and the matcher we assured correctness of our accelerated fraction of the algorithm. A more precise, exhaustive evaluation is possible after accelerating the SBA as a whole.

### 4.3 Results

For the same reasons above, a fair speedup evaluation is not provided. Since, at the best of our knowledge, no hardware implementations of the algorithm have been published our design, when completed, should be compared against the PyMatching software baseline.

## 5 Conclusions

This work represents the first step towards SBA hardware acceleration. A natural future development will be completing the implementation exploiting the same acceleration strategies used for the flooder component. After that, to further increase performance, a possible improvement would be a custom cache implementation or the use of a RTL cache[7].
Only then it will be possible to have a real total speedup measure and assert if a FPGA-based solution for Quantum Error Correction can be actually used

in practice. However, if this won't be the case, the FPGA-based solution could be a starting point in the design of a pure hardware implementation.

## 6 Acknowlegment

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program (formerly known as the XACC program - Xilinx Adaptive Compute Cluster program)[5]

## References

[1] Giovanni Brignone et al. "Array-Specific Dataflow Caches for High-Level Synthesis of Memory-Intensive Algorithms on FPGAs". In: *IEEE Access* 10 (2022), pp. 118858–118877. DOI: `10.1109/ACCESS.2022.3219868`.

[2] Jack Edmonds. "Paths, Trees, and Flowers". In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467. DOI: `10.4153/CJM-1965-045-4`.

[3] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86.3 (Sept. 2012). ISSN: 1094-1622. DOI: `10.1103/physreva.86.032324`. URL: `http://dx.doi.org/10.1103/PhysRevA.86.032324`.

[4] Oscar Higgott and Craig Gidney. "Sparse Blossom: correcting a million errors per core second with minimum-weight matching". In: *arXiv preprint arXiv:2303.15933* (2023).

[5] Javier Moya et al. *fpgasystems/hacc: ETHZ-HACC 2022.1.* Zenodo. `https://doi.org/10.5281/zenodo.8344513`. Sept. 2023. DOI: `10.5281/zenodo.8340448`.

[6] Marco Venere et al. "Towards the Acceleration of the Sparse Blossom Algorithm for Quantum Error Correction". In: (2023).

[7] Felix Winterstein et al. "Custom-sized caches in application-specific memory hierarchies". In: *2015 International Conference on Field Programmable Technology (FPT).* 2015, pp. 144–151. DOI: `10.1109/FPT.2015.7393141`.