Reproducing and Validating Fat-Tree QRAM: A High-Bandwidth Shared Quantum Memory Architecture

Alex Newsham (amn57), Oliver Fogelin (of284)

1 Introduction

Quantum Random Access Memory (QRAM) is a critical primitive for quantum computing, enabling algorithms to query classical databases in superposition [1]. Many important algorithms—including Grover search [2], HHL for solving linear systems [3], Hamiltonian simulation [4], and quantum machine learning schemes [5]—assume QRAM as an ideal primitive and measure complexity only in terms of query count.

The prevailing architecture for QRAM is bucket-brigade (BB) QRAM [1, 6], which uses a binary tree of quantum routers to achieve $O(\log N)$ query latency for a database of size N using $O(N)$ qubits. However, BB QRAM has a significant limitation in shared-memory settings: a single query occupies all routers from root to leaves, forcing concurrent queries to serialize. For p clients, effective latency grows to $O(p \log N)$, making QRAM a bandwidth bottleneck for parallel quantum workloads.

Fat-Tree QRAM, proposed by Xu et al. [7], addresses this limitation by enabling query-level parallelism while retaining BB's space and error scaling. The architecture duplicates routers along an additional index and interleaves controlled-SWAP operations with SWAP layers according to a scheduling protocol. This allows up to $O(\log N)$ concurrent queries to coexist in different slices of the structure, achieving $O(\log N)$ total latency for $O(\log N)$ queries—giving bandwidth that is asymptotically independent of N.

In this work, we reproduce and validate the core claims of Fat-Tree QRAM at the circuit level. Our contributions are:

1. A Qiskit-based implementation of both BB and Fat-Tree QRAM architectures.

2. An implementation of Algorithm 1 from Xu et al., the query scheduling protocol that enables pipelined execution.

3. Validation of correctness via quantum simulation, comparing pipelined multi-query behaviour against sequential execution.

4. Verification of resource scaling claims, measuring qubit counts, circuit depth, and gate counts as functions of N.

ALEX: INSERT BRIEF SUMMARY OF RESULTS HERE PLS MAYBE

2 Background

2.1 Quantum Random Access Memory

A QRAM provides quantum algorithms with the ability to query a classical database of N entries in superposition. Formally, the QRAM query operation is defined as:

$$\sum_{i=0}^{N-1} \alpha_i \left|i\right\rangle_A \left|0\right\rangle_B \longmapsto \sum_{i=0}^{N-1} \alpha_i \left|i\right\rangle_A \left|x_i\right\rangle_B$$

where $\left|i\right\rangle_A$ is an $n = \log_2 N$ qubit address register prepared in a superposition with amplitudes α_i, $\left|x_i\right\rangle_B$ is a bus register that receives the data, and $\{x_i\}_{i=0}^{N-1}$ is the classical database. In this work, we consider a single-qubit bus (i.e., each $x_i \in \{0, 1\}$).

The key property of QRAM is that it maintains coherence across the superposition: a query to an address in superposition returns the corresponding data values entangled with each address component. Classical RAM cannot operate on addresses in superposition, returning only a single value per query rather than data entangled with each address component. QRAM thus enables quantum algorithms to access data in ways that exploit quantum parallelism.

2.2 Bucket-Brigade QRAM

Bucket-brigade (BB) QRAM [1, 6] arranges $2^n - 1$ quantum routers in a binary tree structure above the classical memory cells. Figure 1(c) shows this H-tree layout for $N = 8$: routers (circles) sit above the data cells $x_0, \ldots, x_7$, with the address and bus qubits entering at the root.

2.2.1 Router Structure and Operations

Each router is modeled as a unit with four logical ports: an *input* port that receives incoming qubits, a *route* register that stores the routing decision, and *left*/*right* output ports connected to child routers. The router operates in one of three states: $\left|0\right\rangle$ (route left), $\left|1\right\rangle$ (route right), or $\left|W\right\rangle$ (wait). The wait state $\left|W\right\rangle$ indicates an inactive router that has not yet received an address bit. In our implementation, we encode $\left|W\right\rangle$ simply as $\left|0\right\rangle$: since all qubits initialize to $|0\rangle$ and the route operation uses controlled-SWAPs, an inactive router with route = $|0\rangle$ will route any incoming qubit leftward by default. This is correct because qubits only reach a router after address bits have been loaded into all ancestor routers, ensuring proper activation order.

Four primitive operations define router behaviour:

- **Load**: Transfer a qubit from an external source (address bus or parent router) into the router's input port. Implemented as a SWAP between the source and input.
- **Store**: Transfer the qubit from the router's input port into its route register, storing the routing decision. Implemented as a SWAP between input and route.
- **Route**: Direct an incoming qubit toward the appropriate child based on the stored route bit. If route = $|0\rangle$, CSWAP the input toward the left child; if route = $|1\rangle$, toward the right child. This is implemented as two CSWAPs with an X gate: `CSWAP(route, input, right); X(route); CSWAP(route, input, left); X(route)`.
- **Transport**: Move qubits between adjacent routers via SWAP operations, enabling bit-level pipelining.

2.2.2 Query Execution

A query proceeds in three phases, illustrated in Figure 1(a) for $N = 8$. During **address loading**, each address bit a_i is loaded into the root and routed down through previously-activated routers to level i, where it is stored. Each router at level i stores address bit a_i and uses it to route subsequent qubits left (if $a_i = 0$) or right (if $a_i = 1$), as shown in Figure 1(b). During **data retrieval**, a bus qubit follows the activated paths to the leaves, where data-controlled X gates flip it according to the classical data values, then returns up the tree. During **address unloading**, address bits are extracted back out through the root, restoring all routers to $|W\rangle$. For an n-bit address, each phase requires $O(n)$ layers, giving $O(\log N)$ total circuit depth.

When the address register is in superposition over multiple addresses, each router at level i enters a superposition of $|0\rangle$ and $|1\rangle$ corresponding to the i-th bit of each address component. Crucially, the routing is *coherent*: each branch of the superposition activates only *one* root-to-leaf path, not all possible paths simultaneously. For example, if the address is $\frac{1}{\sqrt{2}}(|010\rangle + |111\rangle)$, only two paths are activated (addresses 2 and 7), while all other routers remain in $|W\rangle$.

2.2.3 Error Scaling

The coherent routing property is essential for BB QRAM's favourable error scaling [8]. Although the tree contains $O(2^n)$ routers, each amplitude component of a superposition query activates exactly n routers (one per level along a single root-to-leaf path). If each router operation introduces error ε, the total query infidelity scales as $O(\log^2 N \cdot \varepsilon)$, not $O(N \cdot \varepsilon)$. This poly-logarithmic error scaling makes BB QRAM practical for large memories, provided individual router fidelities are sufficiently high.

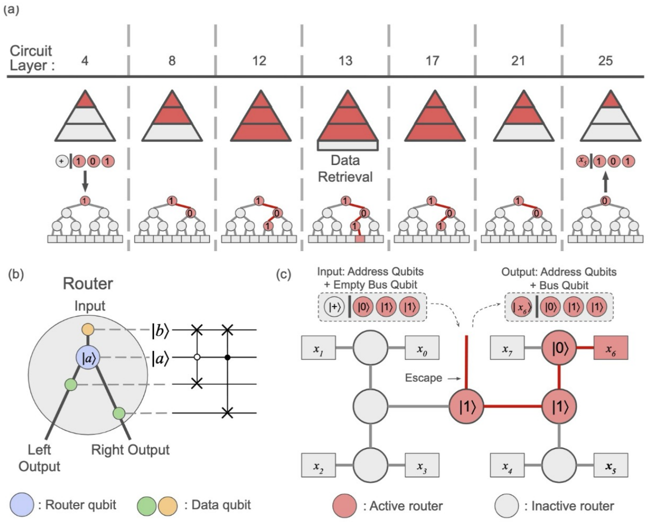

Figure 1: Bucket-Brigade QRAM architecture. (a) Query execution for $N = 8$ showing circuit layers. (b) A single router using CSWAP operations. (c) H-tree layout with active routers (red) after address loading. Figure reproduced from Xu et al. [7].

2.3 Fat-Tree QRAM

The key limitation of BB QRAM in shared-memory settings is that a single query occupies routers along an entire root-to-leaf path, blocking concurrent access. Fat-Tree QRAM [7] overcomes this by *duplicating* routers along a new index k, creating a structure where multiple queries can coexist in different "slices" of the architecture without conflict.

2.3.1 Router Duplication and Structure

Fat-Tree QRAM retains the same binary tree layout as BB QRAM, with 2^i nodes at level i. However, each node at level i contains $(n-i)$ quantum routers instead of one. Routers are indexed by a 3-tuple (i, j, k): $i \in [0, n-1]$ is the level, $j \in [0, 2^i - 1]$ is the node index, and $k \in [0, n-i-1]$ identifies the router copy within node (i, j). Thus:

- Level 0 (root): 1 node with n routers
- Level 1: 2 nodes, each with $n - 1$ routers
- Level $n - 1$ (leaves): 2^{n-1} nodes, each with 1 router

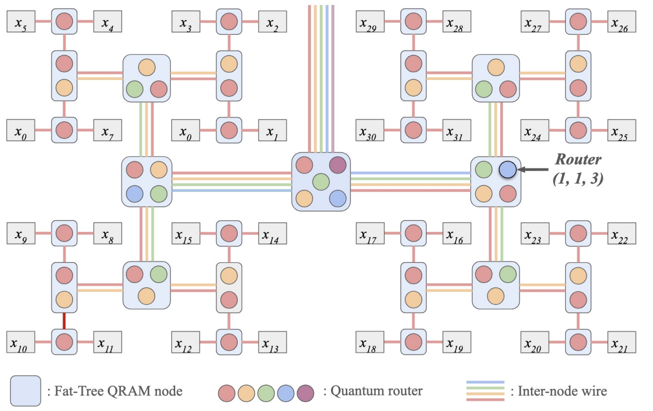

Figure 2: Layout of Fat-Tree QRAM with capacity $N = 32$. Classical data are located at the leaves and internal nodes contain multiplexed quantum routers. Colours indicate connections between routers across the tree. The number of routers per node increases linearly toward the root. Figure reproduced from Xu et al. [7].

Figure 2 illustrates this structure for $N = 32$. Each internal node contains multiple quantum routers (shown as coloured circles), with the number of routers per node decreasing linearly toward the leaves. The total router count is $\sum_{i=0}^{n-1}(n-i)\cdot 2^i = 2N - 2 - n$, approximately twice BB QRAM's $2^n - 1$ routers.

2.3.2 Query Pipelining via SWAP Layers

The key insight enabling parallelism is that queries at different stages of execution can occupy different k-slices simultaneously. Fat-Tree QRAM achieves this through an alternating schedule of **gate steps** and **SWAP layers**:

- **Gate steps**: Perform standard BB QRAM operations (load, route, store) within a fixed k-slice. Each query operates on routers at its current k-level.
- **SWAP-I layers**: Exchange router contents between adjacent k-levels where k is even (i.e., between $k = 0 \leftrightarrow k = 1$, $k = 2 \leftrightarrow k = 3$, etc.).
- **SWAP-II layers**: Exchange router contents between adjacent k-levels where k is odd (i.e., between $k = 1 \leftrightarrow k = 2$, $k = 3 \leftrightarrow k = 4$, etc.).

The execution alternates: gate step → SWAP-I → gate step → SWAP-II → repeat. This pattern ensures that queries "move" through k-space as they progress, vacating lower k-levels for new incoming queries. A new query can be injected once per four-timestep scheduling cycle, allowing up to $O(n) = O(\log N)$ queries to be in flight simultaneously.

2.3.3 Parallelism and Bandwidth

Because queries occupy disjoint k-slices at any given time, they do not conflict on router resources. The result is that $O(\log N)$ independent queries complete in $O(\log N)$ total time steps, giving an effective per-query latency of $O(1)$ in the amortized sense. This represents a bandwidth improvement of $O(\log N)$ over BB QRAM, where p queries require $O(p \log N)$ time.

The trade-off is space: Fat-Tree QRAM requires approximately twice as many routers as BB QRAM. However, for shared-memory workloads with high query contention, this modest space overhead is justified by the dramatic improvement in throughput.

3 Implementation

We implemented both BB QRAM and Fat-Tree QRAM in Python using the Qiskit quantum circuit library. The complete source code is available at `https://github.com/olifog/fat-tree-qram`. This section describes the key challenges encountered during implementation and discusses how we translated the paper's abstract descriptions into working circuits.

3.1 Simulation Scalability

The primary challenge in validating QRAM implementations is simulation scalability. Each router requires four qubits (input, route, left, right), so BB QRAM with n-bit addresses requires $4(2^n - 1)$ router qubits plus address and bus registers. Fat-Tree QRAM approximately doubles this, as each node at level i contains $(n - i)$ router copies. For $n = 4$ (16 memory cells), Fat-Tree QRAM requires approximately 100 qubits—still challenging for statevector simulation on classical hardware.

This constraint forced us to validate correctness on small instances ($n \leq 3$) where exact simulation is tractable, then verify resource scaling claims by examining circuit structure without full simulation. We also focused testing on specific address patterns (all-zeros, all-ones, alternating) rather than exhaustive enumeration.

3.2 Translating Algorithm 1

The paper's scheduling algorithm (Algorithm 1) is described abstractly in terms of "gate steps" and "SWAP layers," but translating this to concrete Qiskit circuits required resolving several ambiguities:

Operation parallelism. The paper does not explicitly specify which operations within a gate step can execute in parallel. We determined that load/store operations at different tree levels can be parallelized, but route operations must respect data dependencies within each query.

Classical data application. The paper states that classical data is applied when queries reach the leaves, but the exact timing relative to SWAP layers was unclear. We found that correct behaviour requires applying data at the SWAP-I layer when n is odd, and at the SWAP-II layer when n is even, corresponding to when queries physically occupy the $k = n - 1$ slice.

Address bit reuse. Since Fat-Tree pipelines multiple queries through shared bus registers, address bits must be reset and reloaded for each new query. The paper's circuit diagrams implied but did not explicitly state this requirement.

3.3 Classical Data Application

The classical database values $\{x_i\}$ must be applied to the bus qubit when it reaches the leaf level. Each leaf router at node index j has its left port corresponding to data index $2j$ and its right port to data index $2j + 1$. The bus qubit, routed to either the left or right port based on the final address bit, picks up the data value via the route operation (which CSWAPs with the appropriate port).

In our BB QRAM implementation, we apply X gates to the leaf router ports twice—once as the bus descends to the leaves and once as it returns. If the data value is 1, the port is flipped; if 0, it remains unchanged. The net effect is an XOR of the data value onto the bus qubit, while the paired X gates cancel out, leaving the router ports in their original state.

For Fat-Tree QRAM, we use a simpler approach: at each SWAP layer when queries occupy the $k = n - 1$ slice (which contains the leaf-level routers), we reset the left/right ports of the relevant routers and conditionally apply X gates based on the classical data values. This occurs at SWAP-I layers when n is odd and at SWAP-II layers when n is even.

3.4 Indexing the Fat-Tree Structure

ALEX: could u pls explain the $(k, \text{tree}, \text{level}, \text{node})$ coordinates into 1D array translation pls

4 Results

4.1 Correctness Validation

4.2 Resource Scaling

4.3 Parallel Query Performance

5 Conclusion

our implementation is awesome

References

[1] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum random access memory". In: *Physical Review Letters* 100.16 (2008), p. 160501.

[2] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM, 1996, pp. 212–219.

[3] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum algorithm for linear systems of equations". In: *Physical Review Letters* 103.15 (2009), p. 150502.

[4] Guang Hao Low and Isaac L. Chuang. "Hamiltonian simulation by qubitization". In: *Quantum* 3 (2019), p. 163.

[5] Jacob Biamonte et al. "Quantum machine learning". In: *Nature* 549.7671 (2017), pp. 195–202.

[6] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Architectures for a quantum random access memory". In: *Physical Review A* 78.5 (2008), p. 052310.

[7] Shifan Xu, Alvin Lu, and Yongshan Ding. "Fat-Tree QRAM: A High-Bandwidth Shared Quantum Random Access Memory for Parallel Queries". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. ACM, 2025. DOI: 10.1145/3676641.3716256.

[8] Srinivasan Arunachalam et al. "On the robustness of bucket brigade quantum RAM". In: *New Journal of Physics* 17.12 (2015), p. 123010.