scientific reports

Check for updates

OPEN

# A quantum random access memory (QRAM) using a polynomial encoding of binary strings

Priyanka Mukhopadhyay

**Quantum algorithms claim significant speedup over their classical counterparts for solving many problems. An important aspect of many of these algorithms is the existence of a quantum oracle, which needs to be implemented efficiently in order to realize the claimed advantages in practice. A quantum random access memory (QRAM) is a promising architecture for realizing these oracles. In this paper we develop a new design for QRAM and implement it with Clifford+T circuit. We focus on optimizing the T-count and T-depth since non-Clifford gates are the most expensive to implement fault-tolerantly in most error correction schemes. Integral to our design is a polynomial encoding of bit strings and so we refer to this design as $\mathbf{QRAM}_{poly}$. Compared to the previous state-of-the-art bucket brigade architecture for QRAM, we achieve an exponential improvement in T-depth, while reducing T-count and keeping the qubit-count same. Specifically, if *N* is the number of memory locations to be queried, then $\mathbf{QRAM}_{poly}$ has T-depth $O(\log\log N)$, T-count $O(N-\log N)$ and uses $O(N)$ logical qubits, while the bucket brigade circuit has T-depth $O(\log N)$, T-count $O(N)$ and uses $O(N)$ qubits. Combining two $\mathbf{QRAM}_{poly}$ we design a quantum look-up-table, $\mathbf{qLUT}_{poly}$, that has T-depth $O(\log\log N)$, T-count $O(\sqrt{N})$ and qubit count $O(\sqrt{N})$. A quantum look-up table (qLUT) or quantum read-only memory (QROM) has restricted functionality than a QRAM. For example, it cannot write into a memory location and the circuit needs to be compiled each time the contents of the memory change. The previous state-of-the-art CSWAP architecture has T-depth $O(\sqrt{N})$, T-count $O(\sqrt{N})$ and qubit count $O(\sqrt{N})$. Thus we achieve a double exponential improvement in T-depth while keeping the T-count and qubit-count asymptotically same. Additionally, with our polynomial encoding of bit strings, we develop a method to optimize the Toffoli-count of circuits, specially those consisting of multi-controlled-NOT gates.**

Quantum computers hold immense promise to outperform classical computers in many applications. Over the years numerous quantum algorithms have been developed that claim speedups over their classical counterparts in various problems, for example, unstructured database search[1], optimization[2], quantum chemistry algorithms[3–7], data processing for machine learning[8–14], cryptography[15,16], image processing[17]. Many of these algorithms require access to oracles in order to fetch classical data and in practice, this is a non-trivial task. It is important to specify the details of implementations of these oracles in order to claim a genuine quantum speedup[18]. Efficient implementation of oracles can reduce the crossover of runtime between classical and quantum advantage from years to days[19].

Till date, the most general-purpose design for the implementation of quantum oracles is a quantum random access memory (QRAM)[20–24], which analogous to a classical random access memory (RAM), returns the element stored in a particular memory location. Specifically, suppose there are $N$ memory locations, each indexed by an integer $i \in \{0, 1, \ldots, N-1\}$ and element $x_i$ is stored in location $i$. Then on input $i$, a classical RAM returns $x_i$. This procedure is called "reading" from the memory. A classical RAM is also able to "write" a particular data $x_i$ into memory location $i$. With a QRAM we are able to query a superposition of addresses. Let $A$ be the input qubit register storing the memory address to be queried and $B$ be the output register. If $|\psi_{in}\rangle$ and $|\psi_{out}\rangle$ are the input and output states, then

Department of Computer Science, University of Toronto, Toronto, ON, Canada. email: mukhopadhyay.priyanka@gmail.com; priyanka.mukhopadhyay@utoronto.ca

$$|\psi_{in}\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle^A |0\rangle^B$$
$$|\psi_{out}\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle^A |x_i\rangle^B. \quad (1)$$

The above equations correspond to the process equivalent to "reading". Like its classical counterpart, a QRAM is also able to "write" into a memory location. In this case, the input state is

$$|\psi'_{in}\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle^A |x_i\rangle^B, \quad (2)$$

and after the operation $x_i$ is XOR-ed into the memory location *i*. The oracles described in many algorithms[25] do require the writing operation. Giovannetti, Lloyd and Maccone introduced the fanout and bucket-brigade architectures for QRAM in their pioneering work in[20,26]. Since then much work has been done to study these designs and improve upon them. Out of the two desgins the bucket-brigade QRAM has become the most popular because it has better noise resilience[27,28] and fault-tolerant resource estimates[29]. Several proposals for the experimental implementations of QRAM have been put forth[20,30–35], each utilizing the bucket brigade architecture. In[36] the authors propose a design implementing an *n*-bit QRAM on hardware nominally supported only on an *m*-bit query, where $m < n$. Over the years there have been proposals for various implementations of QRAM using different techniques, often for specific applications[17,37–53], and thus QRAMs have been used for a wide variety of tasks like neural networks, data processing, quantum communication, image processing, cryptanalysis, quantum simulation, circuit synthesis, state preparation, etc.

A circuit implementing a QRAM needs to be compiled and optimized only once, while the contents of the memory are free to change. But it has the disadvantage of a significant space overhead. A bucket-brigade QRAM for *N* memory locations require *O*(*N*) T gates, *O*(*N*) ancillae and has T-depth $O(\log N)$[29]. In order to reduce the number of ancillae many algorithms use a sequence of multi-controlled-NOT gates, also known as quantum read-only memory (QROM)[4,7,29,54]. This can be implemented with *O*(*N*) T gates, $O(\log N)$ ancillae and *O*(*N*) T-depth. Inspite of a lower qubit count, one disadvantage of a QROM is the exponentially higher T-depth which is not desirable for an efficient fault-tolerant implementation. Another disadvantage of QROM is the fact that we need to know the contents of the memory in advance. Each time the database changes, the circuit needs to be recompiled and optimized. There are other architectures, as in[40,41], that perform queries in $O(N \log N)$ time using $O(\log N)$ qubits.

Many hybrid architectures have been proposed that interpolate between these two extremes and leverage their space-time tradeoff[29,54–59]. Notable among these is the CSWAP architecture[58], which can be thought of as a combination of a QROM and a specific QRAM. It has T-count $O(\sqrt{N})$, number of ancillae $O(\sqrt{N})$ and T-depth $O(\sqrt{N})$. Here we mention that in literature the QRAM, QROM and these hybrid architectures are also used to build quantum look-up-table (qLUT) and so the names are often used interchangeably. These are required to perform restricted tasks. The contents of the table or database are known and this can be leveraged to design circuits with better resource estimates like T-count.

## Our contributions

In this paper we propose an architecture for a QRAM, mainly aimed at reducing the non-Clifford gate complexity of the circuit implementation. We implement our circuits with the fault-tolerant, universal Clifford+T gate set because it implements more unitaries exactly compared to other fault-tolerant, universal gate sets[60–63]. In most error correction schemes the cost of implementing the non-Clifford T gate is significantly higher than the Clifford gates. Thus it is important to optimize the number of T-gates or T-count. It is also important to optimize the T-depth[29,37,64–67], which is related to the running time. A T-depth-1 corresponds to a set of T gates that can be implemented in parallel. We also refer to it as a layer or stage. So a T-depth $\mathcal{T}_d$ for a circuit implies $\mathcal{T}_d$ such stages, where in each stage the T gates are implemented in parallel. The product of T-depth and number of logical qubits is taken as a parameter to measure the rough cost of fault-tolerant implementation in the surface code[29,37,58]. Our contributions in this paper can be summarized in the following points.

(I) We develop a quantum random access memory, which we call $\mathrm{QRAM}_{poly}$ (Section Quantum random access memory ($\mathrm{QRAM}_{poly}$)), with the help of a polynomial encoding of bit strings (Section Polynomial encoding of Boolean strings). We show that $\mathrm{QRAM}_{poly}$ can be implemented with $N - \log N - 1$ Toffoli gates. We can parallelize this circuit, using an additional *O*(*N*) ancillae in order to achieve a Toffoli-depth of $\log \log N$. This implies a T-count of $O(N - \log N - 1)$ and T-depth of $O(\log \log N)$. Thus compared to previous bucket-brigade architecture[29] we achieve an exponential improvement in the T-depth, reduce the T-count, while keeping the number of ancillae nearly the same.

(II) We use two $\mathrm{QRAM}_{poly}$ to design a QROM or qLUT. This is a hybrid architecture and we achieve a T-count of $O(\sqrt{N})$, T-depth $O(\log \log N)$ and ancillae count of $O(\sqrt{N})$. Thus, here we achieve a double exponential improvement in T-depth compared to previous designs[58,59], while keeping the T-count and ancillae count asymptotically similar. We refer to this desgin as $\mathrm{qLUT}_{poly}$ (Section Quantum look-up-table ($\mathrm{qLUT}_{poly}$)). In Table 1 we have summarized and compared the cost of implementation of our $\mathrm{QRAM}_{poly}$ and $\mathrm{qLUT}_{poly}$ with some previous works.

| | T-depth | T-count | #Logical qubits |
|---|---|---|---|
| Bucket-brigade[20,29] | $O(\log N)$ | $O(N)$ | $O(N)$ |
| CSWAP[58] | $O(\sqrt{N})$ | $O(\sqrt{N})$ | $O(\sqrt{N})$ |
| $\mathrm{QRAM}_{poly}$ (This work) | $O(\log\log N)$ | $O(N)$ | $O(N)$ |
| $\mathrm{qLUT}_{poly}$ (This work) | $O(\log\log N)$ | $O(\sqrt{N})$ | $O(\sqrt{N})$ |

**Table 1**. Comparison of T-depth, T-count and number of logical qubits required to implement QRAM and qLUT.

(III) The encoding polynomials that are integral to our constructions of QRAM and qLUT have other applications. For example, in Section Toffoli-count optimization of quantum circuits we describe a procedure (TOFFOLI-OPT-POLY) to optimize the Toffoli-count of circuits. Later we also discuss some other potential applications and hence these polynomials may be of independent interest.

## Organization

In Section Polynomial encoding of Boolean strings we describe a polynomial encoding of bit strings. Using this we design our $\mathrm{QRAM}_{poly}$ in Section Quantum random access memory ($\mathrm{QRAM}_{poly}$). The design of $\mathrm{qLUT}_{poly}$ and a method for circuit optimization has been discussed in Section Other applications of polynomial encoding. Finally we conclude in Section Discussions and conclusion.

## Polynomial encoding of Boolean strings

In this section we describe an encoding where a bit string of length $n$ is represented by a polynomial and then we derive certain properties of the set of $N = 2^n$ polynomials. These attributes will aid in the design of $\mathrm{QRAM}_{poly}$, as explained in later sections.

**Notations :** We use the following notations and conventions. A **polynomial** in $n$ variables comprises of a sum of one or more **monomials**, where each monomial is the product of at most $n$ variables. We say that a monomial has **weight** $k$ if it is the product of $k$ variables. A **constant** is a monomial of weight 0. A polynomial is **linear** if it can be expressed as sum of monomials of weight at most 1. Let $\mathcal{I} \subseteq \{1, 2, \ldots, n\}$ be a subset of indices of the variables $x_1, \ldots, x_n$. We refer to the subscripts as indices. We denote a monomial with variables having indices in $\mathcal{I}$ by $m_{\mathcal{I}}$. That is,

$$m_{\mathcal{I}} = \prod_{j \in \mathcal{I}} x_j. \tag{3}$$

**Encoding polynomial :** Suppose we have an $n$-length bit string - $(b_1, b_2, \ldots, b_n)$, denoted as $\vec{b}$. We encode this bit string into a polynomial in $n$ Boolean variables - $x_1, x_2, \ldots, x_n$, where variable $x_i$ corresponds to bit $b_i$. We assign the following polynomial to each variable $b_i$.

$$b_i \mapsto \frac{1 + (-1)^{b_i}}{2} + x_i := p_{b_i}(x_i) \tag{4}$$

If $b_i = 0$ then $b_i \mapsto \frac{1+1}{2} + x_i = 1 + x_i$ and if $b_i = 1$ then $b_i \mapsto \frac{1-1}{2} + x_i = x_i$. The complete bit string $(b_1, b_2, \ldots, b_n)$ is encoded as follows.

$$(b_1, b_2, \ldots, b_n) \mapsto \prod_{i=1}^{n} \left( \frac{1 + (-1)^{b_i}}{2} + x_i \right) = \prod_{i=1}^{n} p_{b_i}(x_i) := p_{\vec{b}}(x_1, \ldots, x_n) \tag{5}$$

Now we prove some properties of the encoding polynomials.

**Lemma 1** *Suppose we have n bits - $b_1, b_2, \ldots, b_n$ and we associate a variable $x_i$ to each bit $b_i$. Consider the $2^n$ encoding polynomials $\{p_{\vec{b}}(x_1, \ldots, x_n)\}$ corresponding to the $2^n$ possible n-bit strings $\vec{b} = (b_1, b_2, \ldots, b_n)$, as defined in Eq. (5). Then we have*

$$p_{\vec{b}}(b'_1, b'_2, \ldots, b'_n) \equiv \delta_{\vec{b},\vec{b'}} \mod 2, \qquad \text{where} \quad \vec{b'} = (b'_1, b'_2, \ldots, b'_n),$$

*implying $p_{\vec{b}}(b'_1, b'_2, \ldots, b'_n) \equiv 1 \mod 2$ if and only if $\vec{b} = \vec{b'}$ or $b_j = b'_j$ for each $j = 1, \ldots, n$. Else, it is $0 \mod 2$.*

**Proof** By definition of the encoding in Eq. (4), $p_{b_i}(x_i) = 1 + x_i$ when $b_i = 0$ and $p_{b_i}(x_i) = x_i$ when $b_i = 1$. Thus, $p_{b_i}(b_i) = 1$ and since $p_{\vec{b}}(x_1, \ldots, x_n) = \prod_{i=1}^{n} p_{b_i}(x_i)$, so $p_{\vec{b}}(b_1, \ldots, b_n) \equiv 1 \mod 2$.

Again, if $b'_i \neq b_i$ then $p_{b_i}(b'_i) = 2$ or 0. So $p_{\vec{b}}(b'_1, \ldots, b'_n) \equiv 0 \mod 2$, whenever $\vec{b'} \neq \vec{b}$.
This proves the lemma. □

**Lemma 2** *Let $p_{\vec{b}}(x_1, \ldots, x_n)$ be the encoding polynomial corresponding to the bit string $\vec{b} = (b_1, \ldots, b_n)$, as defined in Eq. (5). Assume that k of the bits i.e. $b_{i_1}, \ldots, b_{i_k}$ are 1 and the rest 0. Then,*

$$p_{\vec{b}}(x_1, \ldots, x_n) = \sum_{\mathcal{I}' : \mathcal{I}' \supseteq \mathcal{I}} m_{\mathcal{I}'}, \qquad \text{where} \qquad \mathcal{I} = \{i_1, \ldots, i_k\}.$$

**Proof** Let $\overline{\mathcal{I}} = \{1, \ldots, n\} \setminus \mathcal{I}$ be the complement set of $\mathcal{I}$. All additions and multiplications are commutative. By definition,

$$\begin{aligned} p_{\vec{b}}(x_1, \ldots, x_n) &= \left( \prod_{j : j \in \mathcal{I}} x_j \right) \left( \prod_{\ell : \ell \in \overline{\mathcal{I}}} (1 + x_\ell) \right) \\ &= \left( \prod_{j : j \in \mathcal{I}} x_j \right) \left( 1 + \sum_{\ell \in \overline{\mathcal{I}}} x_\ell + \sum_{\substack{\ell_1 \neq \ell_2 \\ \ell_1, \ell_2 \in \overline{\mathcal{I}}}} x_{\ell_1} x_{\ell_2} + \cdots + \prod_{\ell \in \overline{\mathcal{I}}} x_\ell \right), \end{aligned} \tag{6}$$

which clearly proves the lemma. □
We have the following corollaries.

**Corollary 3** *Let $p_{\vec{b}}(x_1, \ldots, x_n)$ be the encoding polynomial corresponding to the bit string $\vec{b} = (b_1, \ldots, b_n)$, as defined in Eq. (5).* Let $\mathcal{I}_1$ *be the subset of indices of the bits in $\vec{b}$ that have value 1. Given any subset of indices $\mathcal{I} \subseteq \{1, \ldots, n\}$, the monomial $m_{\mathcal{I}}$ appears as a summand in $p_{\vec{b}}(x_1, \ldots, x_n)$ if and only if $\mathcal{I}_1 \subseteq \mathcal{I}$.*

**Corollary 4**

$$\prod_{j=1}^{n} (1 + x_j) = 1 + \sum_{j=1}^{k} x_j + \sum_{j \neq k} x_j x_k + \sum_{j \neq k \neq \ell} x_j x_k x_\ell + \cdots + \prod_{j=1}^{n} x_j.$$

**Corollary 5** *Each encoding polynomial $p_{\vec{b}}(x_1, \ldots, x_n)$, defined in Eq. (5), has exactly one summand monomial of minimum weight. Specifically, let $\mathcal{I}_1$ be the subset of indices of the bits in $\vec{b}$ that have value 1. Then the minimum weight monomial is*

$$m_{\mathcal{I}_1} = \prod_{j \in \mathcal{I}_1} x_j.$$

*Also, it follows that each encoding polynomial has a unique minimum weight monomial.*

**New labeling :** Thus we can label each encoding polynomial $p_{\vec{b}}(x_1, \ldots, x_n)$ by $p_{m_{\mathcal{I}}}(x_1, \ldots, x_n)$, where $m_{\mathcal{I}} = \prod_{j \in \mathcal{I}} x_j$ is the minimum weight monomial and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the subset of indices of the bits in $\vec{b}$ that have value 1.

**Theorem 6** *Let $p_{\vec{b}}(x_1, \ldots, x_n)$ be the encoding polynomial corresponding to a bit string $\vec{b} = (b_1, \ldots, b_n)$, as defined in Eq. (5). Suppose $\mathcal{I}_1 \subseteq \{1, \ldots, n\}$ is the subset of indices of the bits in $\vec{b}$ that have value 1. Then,*

$$p_{\vec{b}}(x_1, \ldots, x_n) = p_{m_{\mathcal{I}_1}}(x_1, \ldots, x_n) = m_{\mathcal{I}_1} + \bigoplus_{\mathcal{I}' : \mathcal{I}_1 \subset \mathcal{I}'} p_{m_{\mathcal{I}'}}.$$

*In the above by XOR we mean that the coefficients of same monomials are added modulo 2.*

**Proof** From Corollary 5, $m_{\mathcal{I}_1}$ appears as a summand in $p_{\vec{b}}(x_1, \ldots, x_n)$. Let $|\mathcal{I}_1| = w$. We need to prove that any monomial $m_{\mathcal{I}'}$ such that $\mathcal{I}' \supset \mathcal{I}_1$ will be added odd number of times.

Consider the sets $\mathcal{I}_{2j} = \mathcal{I}_1 \bigcup \{j\}$ such that $j \notin \mathcal{I}_1$. If we add polynomials $p_{m_{\mathcal{I}_{2j}}}$ which have $m_{\mathcal{I}_{2j}}$ as the minimum weight monomial then each of these monomials of weight $w + 1$ gets added only once.

Consider the sets $\mathcal{I}_{3jk} = \mathcal{I}_1 \bigcup \{j, k\}$ such that $j, k \notin \mathcal{I}_1$. Now for each pair of indices $j, k \notin \mathcal{I}_1$ we have $\mathcal{I}_{3jk} = \mathcal{I}_{2j} \bigcup \{k\} = \mathcal{I}_{2k} \bigcup \{j\}$. From Lemma 2, the monomial $m_{\mathcal{I}_{3jk}}$ appears as a summand in $p_{m_{\mathcal{I}_{2j}}}$ and $p_{m_{\mathcal{I}_{2k}}}$. It also appears as the minimum weight monomial in $p_{m_{\mathcal{I}_{3jk}}}$. From Corollary 3 it cannot appear in any other polynomial $p_{m_{\mathcal{I}'}}$ where $\mathcal{I}' \supset \mathcal{I}_1$. Thus monomials of the form $m_{\mathcal{I}_{3jk}}$ with weight $w + 2$ gets added odd number of times.

| $\vec{b}$ | $p_{\vec{b}}(x_1, x_2, x_3)$ | |
|---|---|---|
| 000 | $1 + x_1 + x_2 + x_3 + x_1x_2 + x_2x_3 + x_1x_3 + x_1x_2x_3$ | $p_1$ |
| 001 | $x_3 + x_1x_3 + x_2x_3 + x_1x_2x_3$ | $p_{x_3}$ |
| 010 | $x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3$ | $p_{x_2}$ |
| 011 | $x_2x_3 + x_1x_2x_3$ | $p_{x_2x_3}$ |
| 100 | $x_1 + x_1x_2 + x_1x_3 + x_1x_2x_3$ | $p_{x_1}$ |
| 101 | $x_1x_3 + x_1x_2x_3$ | $p_{x_1x_3}$ |
| 110 | $x_1x_2 + x_1x_2x_3$ | $p_{x_1x_2}$ |
| 111 | $x_1x_2x_3$ | $p_{x_1x_2x_3}$ |

**Table 2**. Encoding polynomials for 3 bit strings. In the last column an alternate labeling has been mentioned where each polynomial is indexed by its unique minimum weight monomial.

| $\vec{b}$ | $p_{\vec{b}}(x_1, x_2, x_3, x_4)$ | |
|---|---|---|
| 0000 | $1 + x_1 + x_2 + x_3 + +x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_1$ |
| 0001 | $x_4 + x_1x_4 + x_2x_4 + x_3x_4 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_4}$ |
| 0010 | $x_3 + x_1x_3 + x_2x_3 + x_3x_4 + x_1x_2x_3 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_3}$ |
| 0011 | $x_3x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_3x_4}$ |
| 0100 | $x_2 + x_1x_2 + x_2x_3 + x_2x_4 + x_1x_2x_3 + x_1x_2x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_2}$ |
| 0101 | $x_2x_4 + x_1x_2x_4 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_2x_4}$ |
| 0110 | $x_2x_3 + x_1x_2x_3 + x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_2x_3}$ |
| 0111 | $x_2x_3x_4 + x_1x_2x_3x_4$ | $p_{x_2x_3x_4}$ |
| 1000 | $x_1 + x_1x_2 + x_1x_3 + x_1x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_1x_2x_3x_4$ | $p_{x_1}$ |
| 1001 | $x_1x_4 + x_1x_2x_4 + x_1x_3x_4 + x_1x_2x_3x_4$ | $p_{x_1x_4}$ |
| 1010 | $x_1x_3 + x_1x_2x_3 + x_1x_3x_4 + x_1x_2x_3x_4$ | $p_{x_1x_3}$ |
| 1011 | $x_1x_3x_4 + x_1x_2x_3x_4$ | $p_{x_1x_3x_4}$ |
| 1100 | $x_1x_2 + x_1x_2x_3 + x_1x_2x_4 + x_1x_2x_3x_4$ | $p_{x_1x_2}$ |
| 1101 | $x_1x_2x_4 + x_1x_2x_3x_4$ | $p_{x_1x_2x_4}$ |
| 1110 | $x_1x_2x_3 + x_1x_2x_3x_4$ | $p_{x_1x_2x_3}$ |
| 1111 | $x_1x_2x_3x_4$ | $p_{x_1x_2x_3x_4}$ |

**Table 3**. Encoding polynomials for 4 bit strings. In the last column an alternate labeling has been mentioned where each polynomial is indexed by its unique minimum weight monomial.

Similarly we can generalize this argument to monomials of weight $w' > w$. Consider the index set $\mathcal{I}_{w'} = \mathcal{I}_1 \bigcup \mathcal{I}_2$ such that $|\mathcal{I}_2| = w' - w$. Thus the monomial $m_{\mathcal{I}_{w'}}$ has weight $w'$. From Lemma 2 and Corollary 3 this monomial appears as a summand in all polynomials of the form $p_{m_{\mathcal{I}''}}$, where

$$\mathcal{I}_1 \subset \mathcal{I}'' \subseteq \mathcal{I}_{w'}. \tag{7}$$

Number of subsets of weight $w + \ell$ such that they satisfy the subset relation in Eq. (7) is $\begin{pmatrix} w' - w \\ \ell \end{pmatrix}$. Here $\ell$ varies from 1 to $w' - w$. Thus number of times $m_{\mathcal{I}_{w'}}$ gets added is

$$\sum_{\ell=1}^{w'-w} \begin{pmatrix} w' - w \\ \ell \end{pmatrix} = \sum_{\ell=0}^{w'-w} \begin{pmatrix} w' - w \\ \ell \end{pmatrix} - 1 = 2^{w'-w} - 1 \equiv 1 \mod 2.$$

This proves the theorem. □

In Tables 2 and 3 we have enlisted all the encoding polynomials for 3 and 4-bit strings. We have also specified the alternate labeling of each polynomial, that is, indexed by its unique minimum weight monomial. The various properties proved in this section can be verified from these tables.

## Quantum random access memory ($\mathrm{QRAM}_{poly}$)

In this section we describe the construction of $\mathrm{QRAM}_{poly}$ using the polynomial encoding of bit strings, discussed in the previous Section Polynomial encoding of Boolean strings. We implement the circuits using Clifford+T gate set, as discussed earlier in Section Our contributions.

**Some definitions :** Here we briefly recap the following definitions. The **T-count** of a circuit is the number of T-gates in it. The **Toffoli-count** of a circuit is the number of Toffoli-gates in it. Let $U$ be the unitary implemented

by a circuit. Assume $U$ can be expressed as a product of $\mathcal{T}_d$ unitaries, i.e. $U = \prod_{j=1}^{\mathcal{T}_d} U_j$, where each $U_j$ is such that the T or $T^\dagger$ gates appearing in its circuit can be implemented in parallel. We call $\mathcal{T}_d$ as the **T-depth** of the circuit for $U$. Each $U_j$ has T-depth 1 circuit. We can define the **Toffoli-depth** of a circuit in an analogous manner.

Suppose we have $N = 2^n$ memory locations, each specified or indexed by an $n$-bit string $\vec{b} = (b_1, \ldots, b_n)$, which is its **address**. We have $n$ input qubits $\{q_1, \ldots, q_n\}$, whose state selects a memory location. We call these **address qubits**. The main difference between a qubit and a bit is the fact that the former can be in a superposition of both the $|0\rangle$ and $|1\rangle$ states, while the latter can either have state (or value) 0 or 1. Thus state of qubits $(q_1, \ldots, q_n)$ can be a superposition of bit strings $(b_1, \ldots, b_n)$, each specifying a particular memory location, say $M_{\vec{b}}$. We can encode each bit string with the encoding polynomial (Eq. 5) described in Section Polynomial encoding of Boolean strings. It follows that each memory location $M_{\vec{b}}$ is associated with a polynomial $p_{\vec{b}}(x_1, \ldots, x_n)$ in $n$ variables $\{x_1, \ldots, x_n\}$, uniquely determined by $\vec{b}$. We can alternatively call this its **polynomial address**.

We first describe how one particular memory location with address $\vec{b} = (b_1, \ldots, b_n)$ is queried. That is, the state of the address qubit $q_j = b_j$, for each $j = 1, \ldots, n$. Then it is straightforward to understand the operation of the $\mathrm{QRAM}_{poly}$ circuit, when a superposition of memory locations are queried. An illustration of 3-qubit $\mathrm{QRAM}_{poly}$ has been shown in Fig. 2a.

We allocate $N$ ancillae $a_0, \ldots, a_{N-1}$, such that $a_j$ implements the encoding polynomial of $j$ (in the binary form). Each of these ancilla are initialized in state $|0\rangle$. Here we mention that for the remaining part of the paper we use either integers or their binary representation for indexing. This is for convenience and it should be clear from the context. From Lemma 1 we know that for each of the $N$ possible bit strings only one of these ancilla flips to $|1\rangle$ and it is uniquely determined by the bit string. Thus these ancillae can be used to select memory locations. Each of the input qubits is assigned a variable. A Toffoli can be used to multiply two monomials because it operates as follows, with input states $|x\rangle, |y\rangle, |0\rangle$.

$$TOFFOLI|x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|xy\rangle \tag{8}$$

We perform the following steps. In Fig. 1 we have shown a flowchart depicting these steps.

**Step 1.** **Computing monomials :** We implement the monomials using CNOTs and Toffolis. This can be done by multiplying lower weight monomials, using Toffolis. Each monomial is stored in a specific ancilla. Suppose an ancilla $a_j$ is intended to select memory location $M_j$. Let $m_{\mathcal{I}}$ is the minimum weight monomial of the encoding polynomial corresponding to the binary representation of $j$. Then $m_{\mathcal{I}}$ is stored in $a_j$. By Corollary 5, each monomial gets stored in distinct and uniquely determined ancilla.

**Step 2.** **Computing encoding polynomials :** Using CNOTs we XOR the monomials and implement the encoding polynomials, as stated in Theorem 6. That is, after this step ancilla $a_j$ stores $p_{\vec{b}}(x_1, \ldots, x_n)$, where $\vec{b}$ is the binary representation of $j$. So it can be used to select the memory location $M_j$.

**Step 3.** **Select and compute :** Using a Toffoli controlled on $a_j$ and $M_j$ we copy the memory content onto the output bus (Fig. 2b). This is equivalent to "reading" from the memory. For "writing" into the memory we reverse the control and target at the output bus and $M_j$.

**Step 4.** **Making the operations reversible :** To obtain a fully reversible $\mathrm{QRAM}_{poly}$, after the reading or writing operation we implement the circuit, as described in steps 1 and 2, in the reverse order. This is equivalent to uncomputation.

As an example, consider the 3-qubit $\mathrm{QRAM}_{poly}$ shown in Fig. 2. Apart from the 3 input qubits $q_1, q_2, q_3$ there are $N = 2^3 = 8$ ancillae (Fig. 2a), each intended to select a memory location. We label these ancillae as $a_{000}, a_{001}, \ldots, a_{111}$ and the corresponding memory locations as $M_{000}, M_{001}, \ldots, M_{111}$, respectively. For

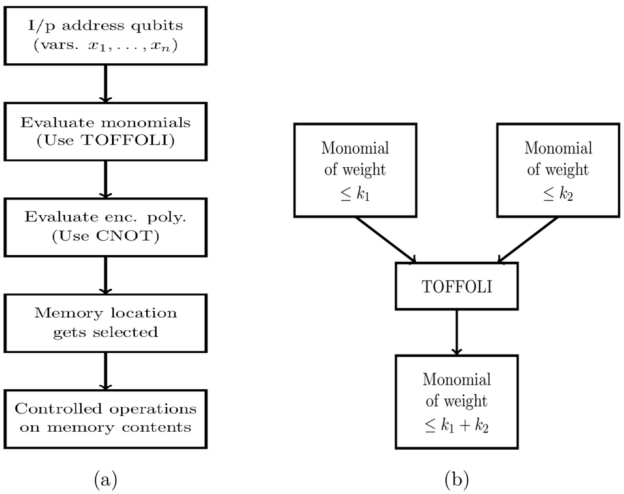


**Fig. 1**. (**a**) A flowchart showing the procedures in $\mathrm{QRAM}_{poly}$. In the figure "I/p" and "enc. poly." represents "Input" and "encoding polynomials", respectively. (**b**) The procedure of evaluating monomials using Toffoli.

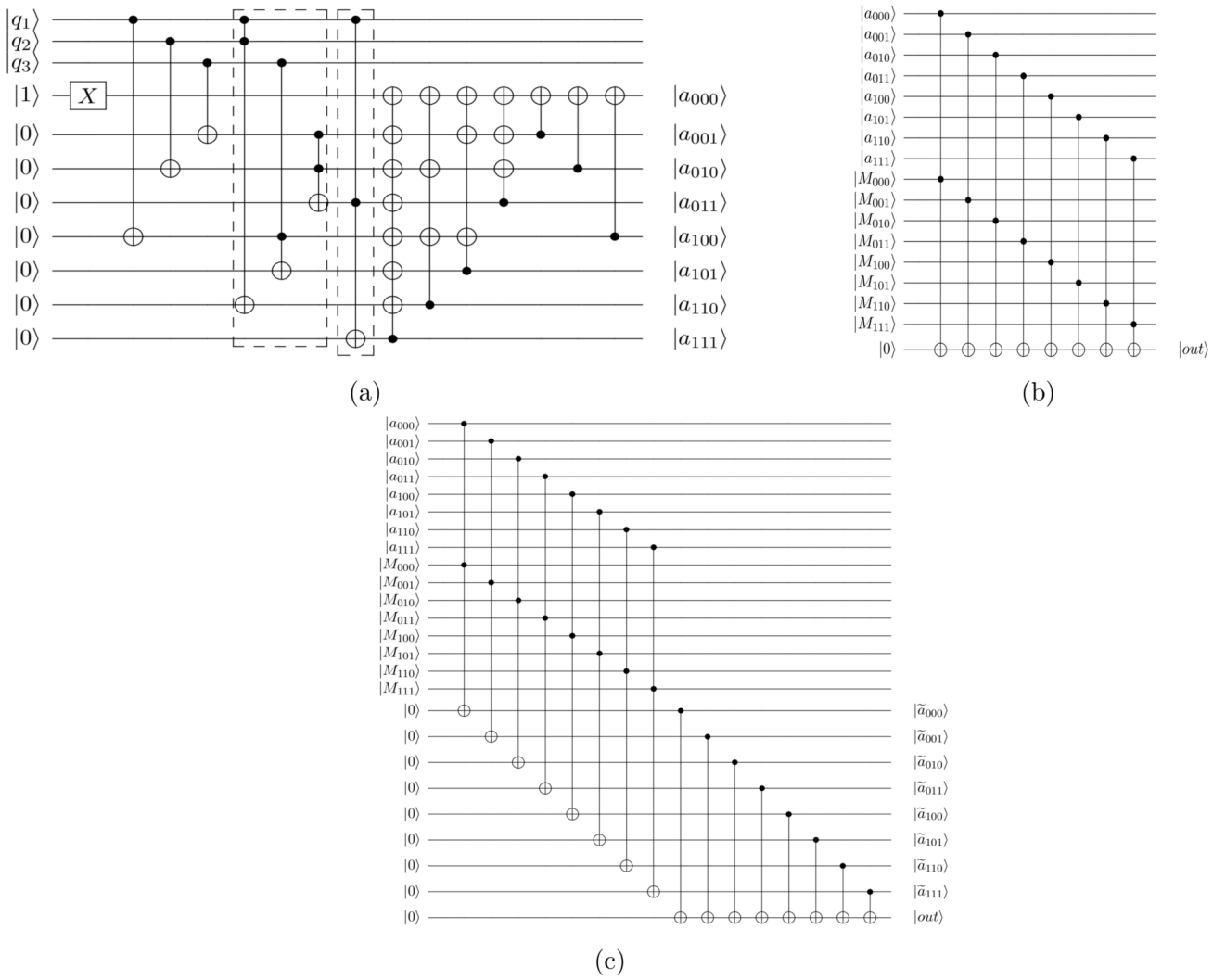


**Fig. 2**. $\mathrm{QRAM}_{poly}$ on 3 qubits. (**a**) The circuit shows the computation of the encoding polynomials and their storage in specific ancilla. These ancillae are used to select memory locations for further operations like reading, writing, assigning phase, etc. Each dotted box corresponds to Toffoli-depth 1. Thus the circuit has Toffoli-depth 2. (**b**) The circuit shows the reading operation where contents of one of the memory location, controlled on an ancilla, is copied to the output bus. (**c**) A parallelized version of the circuit in (b). The Toffoli-depth in this case is 1.

simplicity, we assume each memory location has 1 qubit. Now in each ancilla $a_{\vec{b}}$ we want to implement the polynomial $p_{\vec{b}}$ (Table 2), which is a sum of monomials in variables $x_1, x_2, x_3$, assigned to input qubits $q_1, q_2, q_3$ , respectively. First, we use 3 CNOTs to get the 3 minimum weight monomials of weight 1, that is, $x_1, x_2, x_3$ and store them in $a_{100}, a_{010}, a_{001}$, respectively. Next, using Toffolis we compute monomials of weight 2, that is, $x_1x_2, x_2x_3, x_1x_3$ and store these in ancillae $a_{110}, a_{011}, a_{101}$, respectively. After that we again use Toffoli to multiply monomials of weight 1 and 2, obtaining $x_1x_2x_3$, and store it in $a_{111}$. We observe (refer Table 2) that if a monomial $m_{\mathcal{I}}$ is the minimum weight monomial of an encoding polynomial $p_{\vec{b}}$ then it is stored in ancilla $a_{\vec{b}}$ . This completes Step 1.

Then, using CNOTs we XOR the monomials and obtain the encoding polynomials in corresponding ancilla, as stated in Theorem 6. This implies we first compute encoding polynomials whose minimum weight monomial has highest weight, then we compute those polynomials whose minimum weight monomial has the second highest weight and so on. One way of doing this step is to XOR $a_{\mathcal{I}}$ (storing $m_{\mathcal{I}}$) with each $a_{\mathcal{I}'}$ (storing $m_{\mathcal{I}'}$) such that $\mathcal{I}' \subset \mathcal{I}$. We start from $\mathcal{I}$ with highest cardinality $n$, then sets of indices with second highest cardinality $n-1$ and so on. For example, in Fig. 2a the highest weight monomial is $x_1x_2x_3$ and after step 1, $p_{x_1x_2x_3}$ is already computed in $a_{111}$. We XOR $a_{111}$ with all other ancillae because $\{1,2,3\}$ is a superset of the set of indices of all other monomials. This also computes the polynomials $p_{x_1x_2}, p_{x_2x_3}$ and $p_{x_1x_3}$ in $a_{110}, a_{011}$ and $a_{101}$, respectively. Next, we XOR $a_{\mathcal{I}}$ with each $a_{\mathcal{I}'}$ such that $|\mathcal{I}| = 2$ and $\mathcal{I}' \subset \mathcal{I}$. So, $a_{110}$ is XORed with $a_{100}$ , $a_{010}$ and $a_{000}$. Similarly, we XOR $a_{011}$ and $a_{101}$ with 3 other ancillae (each). This completes the computation of polynomials $p_{x_1}, p_{x_2}$ and $p_{x_3}$ in $a_{100}, a_{010}$ and $a_{001}$, respectively. Finally, we XOR $a_{100}, a_{010}$ and $a_{001}$ with

$a_{000}$. We use X gate on $a_{000}$ to add 1. This completes the computation of polynomial $p_1$. This also completes Step 2.

In Step 3 if we want to read from the memory then we use the circuit in Fig. 2b. Here controlled on each ancilla a memory location is copied to the output bus $|out\rangle$.

## Illustration : application of $\mathbf{QRAM}_{poly}$ in Grover's algorithm

To further illustrate the application of $\mathrm{QRAM}_{poly}$ we consider the Grover's algorithm[1], which gives a thoeretical quadratic speedup over classical search algorithms in an unstructured database. Suppose there are *N* items in the database. The Grover's algorithm consists of the following steps. First we initialize the system in the state $|\Psi\rangle = |0\rangle^n$. Then we perform a number of iterations of the following procedure.

1. Set all qubits into an equal superposition state $|s\rangle$.

$$\mathrm{H}^{\otimes n}|0\rangle^n = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle = |s\rangle.$$

2. Phase-tag the states that represent the values to be searched.
3. Implement a diffusion operator $U_d = 2|s\rangle\langle s| - \mathbb{I}$ that amplifies the amplitudes for measuring the states that need to be searched.

At the end of all iterations of the algorithm we perform a measurement in the computational basis. The searched items can be be found by identifying the distinct peaks in the distribution of the measured results.

Steps 2 (phase-tagging) and 3 (diffusion operator) correspond to two successively performed reflections, and thus together they perform a rotation in a 2D-plane. Thus in each iteration of the Grover's algorithm the state $|s\rangle$ is rotated closer to a state $|k\rangle$, that represents a value to be searched. After an optimal number of iterations $|s\rangle$ is rotated the closest to $|k\rangle$. Searching one item in an unstructured database with *N* items requires at most $O\left(\sqrt{N}\right)$ iterations. Classically this search can be done in $O(N)$ time complexity. Thus it is possible to achieve a quadatic speed-up, provided each iteration is done efficiently, or simply put, time complexity of each iteration is considerably less than the number of iterations.

Usually, in theoretical analyses of Grover's algorithm we assume the existence of a phase-tagging oracle that performs step 2. This oracle has the following functionality :

$$O|i\rangle = (-1)^{f(i)}|i\rangle \quad \text{with} \quad f(i) = 1 \quad \text{if} \quad i \in \{k\} \quad \text{else} \quad f(i) = 0.$$

An efficient implementation of this oracle is essential in order to achieve the claimed speedup of the Grover's algorithm in practice. Our $\mathrm{QRAM}_{poly}$ can be used to implement this oracle. Specifically, after we compute the encoding polynomials in order to select a memory location (Step 3) we do not require the Toffolis, as shown in Fig. 2b. Instead, we use CZ on each memory location where the control is on the selecting ancilla. In this way, we apply phase on selected memory locations.

We are not going into more detail of an optimal fault-tolerant implementation of Grover's algorithm in order to achieve a practical quantum speed-up as this is a stand-alone research topic[73] and beyond the scope of this paper. But briefly we want to summarize this section by emphasizing that an efficient implementation of the phase-tagging oracle is crucial for the practical speed-up of Grover's algorithm. A QRAM (with proper modification) can be used and faster this QRAM, the better it is. In later sections we give a detail analysis of the cost of fault-tolerant implementation of $\mathrm{QRAM}_{poly}$ (though with read/write operations) with the surface code and show that it is much faster and consumes less number of qubits, compared to other QRAMs. Thus it can also be used to have a faster fault-tolerant implementation of Grover's algorithm.

## Cost of implementation

Now we discuss the cost of implementing a circuit with the Clifford+T gate set. We focus on optimizing the non-Clifford resource, that is, the T-count and T-depth, as discussed earlier in Section Our contributions. We first bound the Toffoli depth and number of compute-uncompute Toffoli pairs. In literature there are different implementations of a Toffoli gate with the Clifford+T gate set, some optimizing its T-count, while there are others that optimize T-depth. The circuit in[68,69] gives the lowest T-count of 4 and has a T-depth of 2. It uses logical AND gadget which does the multiplication. One advantage of this circuit is the fact that if we have a compute-uncompute pair then the uncomputation part does not require any T-gate, but it uses classical measurements. Another circuit is the one in[70] which has a T-count of 7, T-depth of 1 and uses 4 extra ancillae. Depending on whichever parameter we want to optimize, one implementation can be favoured over the other.

We have separated the cost after the computation of the encoding polynomials since the cost of this part can change depending upon the required operations, as discussed in Section Illustration : application of $\mathrm{QRAM}_{poly}$ in Grover's algorithm.

**Number of compute-uncompute Toffoli pairs :** From Theorem 6 we know that we need to implement all monomials of weight $0, 1, 2, \ldots, n$. We do not require any Toffoli to implement monomials of weight 0 and 1. The former can be obtained by applying X gate and the latter are the variables assigned to the input qubits. There are $\binom{n}{2}$ monomials of weight 2. We need these many Toffolis in order to get all monomials of weight

2. Monomials of weight 3 can be obtained by multiplying each weight 2 monomial with a variable. In general, a monomial of weight $k \geq 2$ can be obtained by multiplying any two already calculated monomials of weights $k_1, k_2 < k$, such that $k_1 + k_2 = k$. Thus we require 1 Toffoli to compute each monomial of weight more than 1. We also require equal number of Toffolis in order to uncompute. Hence the number of compute-uncompute pairs of Toffoli we require is

$$\mathcal{T}_c^{of} = \sum_{k=2}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} - n - 1 = 2^n - n - 1 = N - \log_2 N - 1. \tag{9}$$

Additionally, for copying memory contents (Fig. 2b), we require $N\ell$ Toffolis, where $\ell$ is the number of qubits in each memory location or its size. This implies that $\text{QRAM}_{poly}$ has a T-count of $O(N - \log_2 N - 1)$ for computing the encoding polynomials. It requires an additional $O(N\ell)$ T-gates for reading or writing.

**Number of logical qubits :** Apart from the $n = \log_2 N$ qubits containing the input address, we require $N$ ancillae in order to select memory locations. Thus the number of logical qubits, excluding the $N\ell$ memory qubits is

$$\mathcal{Q} = N + \log_2 N. \tag{10}$$

**Number of CNOT pairs :** Suppose an ancilla $a_j$ selects memory location $M_j$. We have already discussed that we need to implement the encoding polynomial of the binary representation of $j$ in $a_j$. From Corollary 5 we know that each encoding polynomial has a unique minimum weight monomial determined by the bit string that it encodes. We can label each ancilla using these minimum weight monomials. Let $\mathcal{I}$ be the set of indices of the bits in the binary representation of $j$, that have value 1. $m_{\mathcal{I}}$ is the corresponding minimum weight monomial of the encoding polynomial $p_{\text{bin}(j)}(x_1, \ldots, x_n)$, where $\text{bin}(j)$ is the binary representation of $j$. Then we can alternatively refer to $a_j$ as $a_{\mathcal{I}}$.

Initially, using $n$ CNOTs we store the single weight monomials $x_1, \ldots, x_n$ in $n$ ancillae - $a_{\{1\}}, \ldots, a_{\{n\}}$ , respectively. Then using Toffolis we compute and store monomials of weight greater than 1 in different ancillae. According to Theorem 6 we use CNOTs to add these monomials in order to implement the encoding polynomials. Consider a monomial $m_{\mathcal{I}}$ of weight $k$, i.e. $|\mathcal{I}| = k$ that has been computed in ancilla $a_{\mathcal{I}}$. From Corollary 3 we know that we need to add a CNOT from the ancilla $a_{\mathcal{I}}$ to each ancilla $a_{\mathcal{I}'}$, where $\mathcal{I}' \subset \mathcal{I}$. Now $\mathcal{I}$ has $2^k - 1$ subsets (excluding itself). And there are $\binom{n}{k}$ monomials of weight $k$. We need an equal number of CNOT for uncomputations. Thus total number of CNOT pairs required is

$$\begin{aligned} \mathcal{C} =& n + \sum_{k=1}^{n} \binom{n}{k} (2^k - 1) = n + \sum_{k=0}^{n} \binom{n}{k} 2^k - \binom{n}{0} 2^0 - \sum_{k=0}^{n} \binom{n}{k} + \binom{n}{0} = n + 3^n - 2^n \\ &= \log_2 N + N^{\log_2 3} - N \approx \log_2 N + N^{1.6} - N. \end{aligned} \tag{11}$$

**Toffoli-depth :** In this design Toffolis are required to compute the different monomials that are stored in distinct ancillae. As mentioned, we can compute a monomial of weight $k$ by multiplying two already-computed monomials of weight $k_1, k_2 < k$ such that $k_1 + k_2 = k$. Initially we have $n$ monomials of weight 1 (the inputs) that are also stored in $n$ ancillae. So we have $2n$ monomials and using these we can compute $\frac{2n}{2} = n$ new monomials of higher weight in parallel. After that we can compute $\frac{2n+n}{2} = \frac{3n}{2}$ monomials in parallel using the already available $3n$ monomials. Next, we can compute $\frac{1}{2}\left(2n + \frac{2n}{2} + \frac{1}{2}\left(2n + \frac{2n}{2}\right)\right)$ monomials in parallel using the available monomials. Roughly, we can compute $O(n)$ monomials in parallel in each round. Since we need to compute $2^n - n - 1$ monomials of weight greater than 1, so Toffoli-depth is

$$\mathcal{T}_d^{of} \in O\left(\frac{2^n - n - 1}{n}\right) \in O\left(\frac{N - \log_2 N - 1}{\log_2 N}\right). \tag{12}$$

## Parallelizing the multiplications : lower T-depth

We observe that in this design the non-Clifford Toffolis are required to compute monomials of weight at most $n$. Since we store the monomials in different qubits so we can compute in parallel monomials of weight $k$ using monomials of weight $1, \ldots, \frac{k}{2}$. Thus the maximum weight of any monomial that can be computed in parallel is double the maximum weight of any monomial computed in the previous parallel stage (Each stage is Toffoli-depth 1). Hence, all the monomials can be computed with Toffoli-depth

$$\mathcal{T}_{d,par}^{of} = \log_2 n = \log_2 \log_2 N. \tag{13}$$

In Fig. 3 we have shown a parallel implementation of a 4-qubit QRAM.

The set of $N\ell$ Toffolis required to read or write from the memory can be parallelized to have Toffoli-depth 1. A simple method using an additional $N\ell$ number of ancillae, has been shown in Fig. 2c, where $\ell = 1$. During reading, the contents of each memory location is copied to an ancilla. Then using CNOTs the parity of these ancillae is stored on the output bus. As explained before, during writing the control and target of the memory locations and output bus are reversed. In order to parallelize these operations and have Toffoli-depth 1, we can also use the method described in Figure 4 of[29] or in[71], the latter has exponentially less qubit-count.

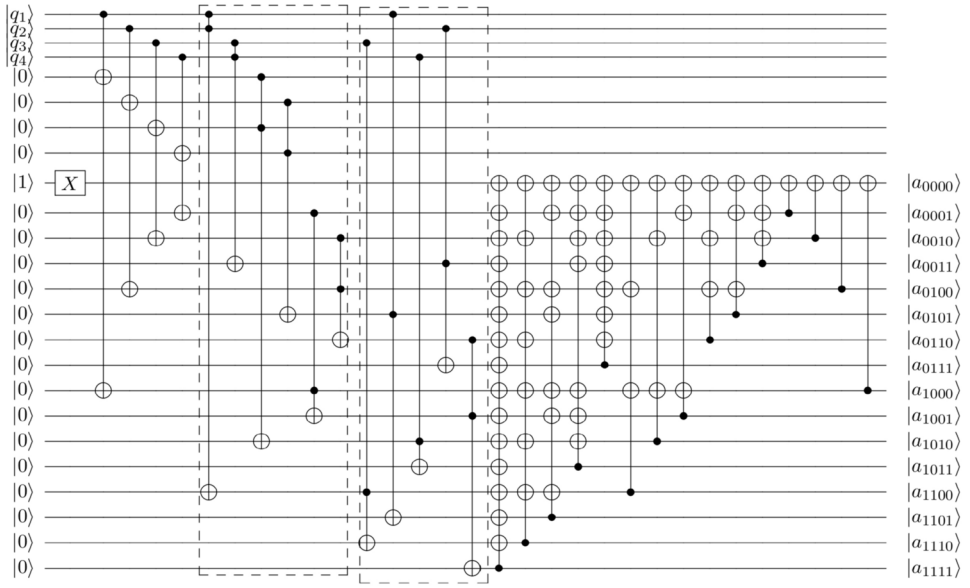


**Fig. 3**. $\mathrm{QRAM}_{poly}$ on 4 qubits. This is a parallelized version, where the Toffoli-depth has been reduced by using extra ancillae. Each dotted box corresponds to Toffoli-depth 1. Thus the circuit has Toffoli-depth 2. We show only till the computation of the encoding polynomials in respective ancillae.

Hence, using either of the implementations in[68,69] or[70], accounting for both the computation and uncomputation part, we achieve a T-depth of $2\log_2\log_2 N$, excluding the read/write operation.

**Extra ancillae to reduce Toffoli-depth :** In the first step we compute monomials of weight 2, using monomials of weight 1. We store $n+n$ weight-1 monomials in $2n$ distinct qubits. Using these we can compute $n$ weight-2 monomials in parallel. We can compute the remaining $\binom{n}{2} - n$ weight-2 monomials by copying the inputs in different ancillae. So we require

$$2\left(\binom{n}{2} - n\right)$$

extra ancillae. We reuse ancillae. In the next step (Toffoli-depth 2) we compute monomials of weight 3 and 4. We can already compute $\frac{1}{2}\left(2n + \binom{n}{2}\right)$ monomials using the monomials already stored (no extra ancilla). We can compute the remaining using

$$2\left(\binom{n}{4} + \binom{n}{3}\right) - \left(\binom{n}{2} + 2n\right)$$

extra ancillae. We use the ancillae used in Toffoli-depth-1. So we need not add this number to the previously calculated number. Generalizing, suppose we have computed all monomials of weight at most $k/2$. In the next step we can compute all monomials of weight $k$. We can compute $\frac{1}{2}\left(\binom{n}{k/2} + \cdots + 2n\right)$ monomials using already stored monomials. Remaining can be computed using

$$A_k = 2\left(\binom{n}{k} + \cdots + \binom{n}{k/2+1}\right) - \left(\binom{n}{k/2} + \cdots + \binom{n}{2} + 2n\right)$$

extra ancillae. Number of extra ancillae we require is

$$\max_k A_k \leq 2^n = N. \tag{14}$$

Thus, excluding the memory qubits and ancillae required to parallelize the reading/writing operation, the total number of logical qubits we require is

$$\mathcal{Q}_{par} \leq 2N + \log N. \tag{15}$$

**Extra CNOT to reduce T-depth :** At each step we require CNOTs to copy monomials to extra ancillae and then again we reset. Roughly, we can say that we require $\max_k A_k$ CNOT pairs at each step. Thus number of extra CNOT pairs is at most

$$2^n \log_2 n = N \log\log N.$$

So total number of CNOT pairs is

$$\mathcal{C}_{par} \in O\left(3^n - 2^n + 2^n \log_2 n\right) \in O\left(N^{1.6} + N \log_2 \log_2 N\right). \tag{16}$$

***Remark 3.1*** A monomial of weight $k$ can be computed by $C^k X$, a NOT gate controlled on $k$ qubits. This gate, on input $|x_1\rangle, |x_2\rangle, \ldots, |x_k\rangle, |0\rangle$, returns the product $|x_1 x_2 \ldots x_k\rangle$, as follows.

$$C^k X|x_1\rangle|x_2\rangle \cdots |x_k\rangle|0\rangle \mapsto |x_1\rangle|x_2\rangle \cdots |x_k\rangle|x_1 x_2 \ldots x_k\rangle$$

In principle, we can compute all the necessary $N - \log_2 N - 1$ monomials in parallel by using $\binom{n}{k}$ number of $C^k X$ gates, for each $k = 2, \ldots, n$. We require enough number of ancillae in order to copy the input variables the required number of times. Thus for computing the encoding polynomials the T-depth or Toffoli-depth is determined by the maximum T-depth or Toffoli-depth required to implement any $C^k X$, where $k = 2, \ldots, n$. So this part can improve with the design of better circuits for $C^n X$.

## Comparison with previous work

We compare the resource estimates with the parallelized version of bucket-brigade QRAM[29]. Here we mention that we have compared with the bucket-brigade architecture for the following reasons. First, this has been the most widely studied QRAM and a detailed fault-tolerant resource estimates is available, as in[29]. Second, its applications are more general than other QRAMs, for example, FF-QRAM[40], EQGAN-QRAM[72], PQC-based QRAM[49], which have been designed for specific problems and some of them do not scale well. Third, QRAMs as in[40,49,72–74] use unitaries like controlled rotations, that are approximately implementable by discrete universal gate sets. The non-Clifford cost like T-count or Toffoli-count varies inversely with the precision of synthesis[61–63,75]. Hence, such designs are more expensive to implement fault-tolerantly. Fourth, many works have used the bucket-brigade QRAM as a basic module and designed more intricate architecture with it, most often to achieve some tradeoffs for particular applications or scenario, for example[36]. Thus, we can simply replace this module with our $\mathrm{QRAM}_{poly}$ in order to compare the designs.

We compare the Toffoli-count, Toffoli-depth and number of logical qubits. In this way we do not have to worry about the difference in T-count and T-depth due to different implementations of Toffoli. We do not consider resource estimates for the reading/writing operation since this part can be implemented and optimized in a similar fashion and hence its cost can be regarded the same. Without this, the parallel bucket-brigade circuit in[29] requires $N - 2$ compute-uncompute Toffoli pairs, $2N + \log_2 N$ logical qubits and has Toffoli-depth $\log_2 N$. From Eqs. (9), (13) and (15), this implies the following.

$$\mathcal{R}_{T-depth} = \frac{\text{T-depth in this work}}{\text{T-depth in [29]}} = \frac{\log_2 \log_2 N}{\log_2 N} \tag{17}$$

$$\mathcal{R}_{T-count} = \frac{\text{T-count in this work}}{\text{T-count in [29]}} = \frac{N - \log_2 N - 1}{N} \tag{18}$$

$$\mathcal{R}_{qubits} = \frac{\#\text{Logical qubits in this work}}{\#\text{Logical qubits in [29]}} = \frac{2N}{2N} = 1 \tag{19}$$

Thus we achieve an exponential improvement of T-depth, reduction in T-count, while keeping the number of logical qubits the same.

**Fault-tolerant implementation :** In most error-correction schemes, including the most popular surface code, the cost of implementation of the non-Clifford T gate is much more than the cost of the Clifford gates[64]. In[29] the following has been taken as a rough estimate for the cost of implementation with the surface code.

$$\text{Rough cost } = \log_2 (\text{Logical qubits } \times \text{ T-depth}) \tag{20}$$

Thus from Eqs. (17)-(19) we can say that our $\mathrm{QRAM}_{poly}$ has much less fault-tolerant cost estimates from previous bucket-brigade architecture. So we expect much better performance in terms of running time.

For illustration, we consider an implementation of our $\mathrm{QRAM}_{poly}$ with the surface code, and estimate resource requirements following the procedures described in[64] and[37]. Consider the number of qubits $n = 36$ , which corresponds to 8 GB of classical data. Using logical AND gadgets[69], Toffoli can be implemented with 4 T gates with a T-depth 2 and if there is a compute-uncompute Toffoli pair we do not require any T gate for the uncomputation part. Hence, from Eq. (9), assuming $\ell = 1$, the T-count is

$$\mathcal{T} \leq 4(2N - \log n - 1) = 5.4975 \times 10^{11} \tag{21}$$

and the T-depth is

$$\mathcal{T}_d \leq 2(\lceil \log n \rceil + 1) = 14. \tag{22}$$

Each T gate requires one magic state ($|A_L\rangle$). For the above T-count the output error rate for state distillation should be no greater than

$$p_{out} = \frac{1}{\mathcal{T}} \approx 1.81899^{-12}. \tag{23}$$

Now we use Algorithm 4 in[37] in order to calculate the distance of the surface code. Assuming a magic state injection error rate $p_{in} = 10^{-4}$, a per-gate error rate $p_g = 10^{-5}$, the stated algorithm suggests 2 layers of distillation with distances $d_1 = 10$, $d_2 = 5$. The first stage of distillation consumes 16 logical qubits and the second stage consumes $16 \times 15 = 240$ logical qubits in order to generate a single magic state.

The input states of the logical qubits in the second layer are encoded on a distance $d_2 = 5$ code that uses $N_{p2} = 2.5 \times 1.25 \times d_2^2 \approx 79$ physical qubits per logical qubit. The total footprint of the distillation circuit is then $240 \times N_{p2} = 18750$ physical qubits. This round of distillation is completed in $\sigma_2 = 10d_2 = 50$ surface code cycles.

The first or top layer requires a $d_1 = 10$ surface code, for which a logical qubit takes $N_{p1} = 2.5 \times 1.25 \times d_1^2 \approx 313$ physical qubits. So the total number of physical qubits required is $16 \times N_{p1} = 5000$, with the round of distillation completed in $\sigma_1 = 10d_1 = 100$ surface code cycles.

The concatenated distillation scheme is performed in $\sigma = \sigma_1 + \sigma_2 = 150$ surface code cycles. Since the first or top layer has lower footprint than the second or bottom layer, distillation can potentially be pipelined to produce

$$\frac{150 \times 18750}{100 \times 5000 + 50 \times 18750} \approx 2$$

magic states in parallel. The physical qubits in the second layer is reused. Let $t_{sc} = 200ns$ is a surface code cycle time. Then 2 magic states can be produced every $\sigma \times t_{sc} = 30 \times 10^{-6}$ s. We require $\frac{\mathcal{T}}{\mathcal{T}_d} = \frac{5.4975 \times 10^{11}}{14} \approx 3.9 \times 10^{10}$ magic states per layer or depth of T-gates. We produce these many magic states in parallel for each layer. Due to parallelization the number of physical qubits required is

$$\frac{1}{2} \times 3.9 \times 10^{10} \times 18750 \approx 3.66 \times 10^{14} \tag{24}$$

and the time taken is

$$14 \times 30 \times 10^{-6} s = 4.2 \times 10^{-4} s. \tag{25}$$

In surface code implementation the cost of implementation of a multi-target CNOT is equal to the cost of a single target CNOT[64] and has the same execution time. So we consider a multi-target CNOT as one logical CNOT. Each Toffoli can be implemented with 2 CNOTs, 2 multi-target CNOTs, 1 H and 1 S gate[69]. Thus we can upper bound the number of logical Cliffords by $7N = 7 \times 2^{36} = 4.81 \times 10^{11}$. The overall error rate of the Cliffords should therefore be less than $\frac{1}{7N} \approx 2.08 \times 10^{-12}$. To compute the required distance, we seek the smallest *d* that satisfies the inequality

$$\left(\frac{p_{in}}{0.0125}\right)^{\frac{d+1}{2}} < 2.08 \times 10^{-12}$$

and find this to be $d = 11$. The number of physical qubits required to encode the Cliffords is at most $2 \times 2^{36} \times 2.5 \times 1.25 \times 11^2 \approx 5.197 \times 10^{13}$. Overall, we require approximately $3.66 \times 10^{14} + 5.197 \times 10^{13} \approx 4.18 \times 10^{14}$ physical qubits to encode the complete $\mathrm{QRAM}_{poly}$.

Roughly, we expect to perform $\frac{7N}{2N \times \mathcal{T}_d} = \frac{7}{2 \times 14} \approx 0.25$ logical Clifford operations per qubit per layer of T-depth. Since each logical CNOT takes 2 surface code cycles and we require $\sigma = 150$ surface code cycles per T-depth, so the overall time is dominated by the time for implementing the non-Clifford T-gates. Thus, the time for implementing one memory read/write operation with $\mathrm{QRAM}_{poly}$ is approximately $4.2 \times 10^{-4}$s and utilizes about $4.18 \times 10^{14}$ physical qubits.

The QRAM in[29] requires $1.5 \times 10^{15}$ physical qubits and is implemented in approximately $2.13 \times 10^{-3}$s, using the same surface code parameters and doing a similar analysis. We remark that design considerations can vary. We can further reduce the time by parallelizing the magic state factories even more. This will increase the number of physical qubits.

## Other applications of polynomial encoding

In this section we discuss some applications for the polynomial encoding of bit-strings. First, we describe a quantum look-up-table (qLUT) that can be built using two QRAMs. If we use $\mathrm{QRAM}_{poly}$ that uses the polynomial encoding, then our $\mathrm{qLUT}_{poly}$ has double exponentially less T-depth than previous designs. Second, we descibe a method to optimize the Toffoli-count of circuits consisting of groups of multi-controlled-NOT gates.

### Quantum look-up-table ($\mathrm{qLUT}_{poly}$)

Now we describe a quantum look-up-table formed by combining two QRAMs, as shown in Fig. 4. Suppose we want to build a qLUT with $n$ address bits. That is, there are $N = 2^n$ number of look-up addresses. We divide the address bits into two groups, the first one with $n_1$ bits and the next one with $n_2$ bits. That is, $n = n_1 + n_2$ and assume $N_1 = 2^{n_1}$, $N_2 = 2^{n_2}$. With these we build two QRAMs, $\mathrm{QRAM}_{poly,n_1}$ and $\mathrm{QRAM}_{poly,n_2}$ with $n_1$ and $n_2$ number of address qubits respectively. Thus, $\mathrm{QRAM}_{poly,n_1}$ and $\mathrm{QRAM}_{poly,n_2}$ are capable of addressing $N_1$ and $N_2$ memory locations respectively. And as discussed before we have $N_1$ and $N_2$ number of (selecting) ancillae in each $\mathrm{QRAM}_{poly}$ (respectively) that correspond to these address locations.

Suppose we want to read a look-up address indexed by the bit string $b_0, \ldots, b_{n-1}$, or alternatively by the integer $N' = \sum_{j=0}^{n-1} b_j 2^j$. Now,

$$N' = \sum_{j=0}^{n-1} b_j 2^j = \sum_{j=0}^{n_2-1} b_j 2^j + \sum_{j=n_2}^{n-1} b_j 2^j = \sum_{j=0}^{n_2-1} b_j 2^j + 2^{n_2} \left( \sum_{j=n_2}^{n-1} b_j 2^{j-n_2} \right)$$
$$= \sum_{j=0}^{n_2-1} b_j 2^j + N_2 \left( \sum_{j'=0}^{n-n_2-1} b_{j'+n_2} 2^{j'} \right) = \sum_{j=0}^{n_2-1} b_j 2^j + N_2 \left( \sum_{j'=0}^{n_1-1} b_{j'+n_2} 2^{j'} \right) = N_2' + N_2 N_1'.$$

With the first QRAM i.e. $\mathrm{QRAM}_{poly,n_1}$ we select all address locations with first $n_1$ bits as $b_0, \ldots, b_{n_1-1}$. From the previous equation we know that there are $N_2$ such locations and we copy their contents in separate registers. With the next QRAM i.e. $\mathrm{QRAM}_{poly,n_2}$ we select an address among these copied addresses. Specifically, it selects an address with the last $n_2$ bits as $b_{n_1}, \ldots, b_n$. Hence finally a memory location with address $(b_0, \ldots, b_n)$ gets selected. After this, using Toffolis we copy the contents of this memory location into the output bus.

Let each look-up address has $\ell$ bits. So after an address gets selected we require $N_2\ell$ Toffolis in order to compute the parity into the output bus. We can parallelize these Toffolis with additional $N_2\ell$ ancillae, as shown in Fig. 2c and explained in Section Parallelizing the multiplications : lower T-depth. $\mathrm{QRAM}_{poly,n_1}$ and $\mathrm{QRAM}_{poly,n_2}$ can be implemented in parallel. Further, if we use the parallelized versions of these QRAMs then we have the following.

$$\text{Toffoli-depth} = \max\{\log_2 n_1, \log_2 n_2\} + 1$$
$$\text{Toffoli-count} = (N_1 - n_1 - 1) + (N_2 - n_2 - 1) + \ell N_2 = N_1 + (\ell + 1)N_2 - n - 2$$
$$\#\text{Ancillae} = 2N_1 + 2N_2 + \ell N_2 + \ell N_2 = 2(N_1 + (\ell + 1)N_2)$$

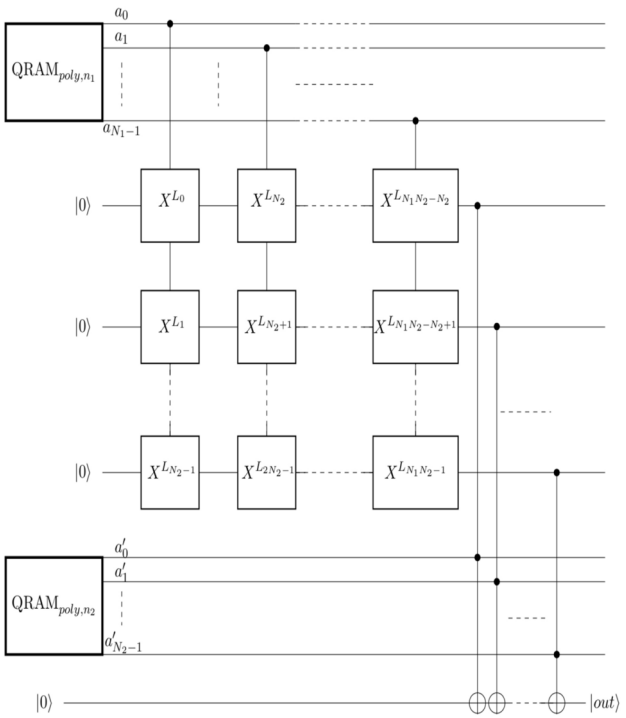


**Fig. 4**. A quantum look-up table (qLUT), using $n_1$-bit and $n_2$-bit QRAM in parallel. For $\mathrm{qLUT}_{poly}$ we use two $\mathrm{QRAM}_{poly}$, one addressing $n_1$-bit locations and another one addressing $n_2$-bit locations.

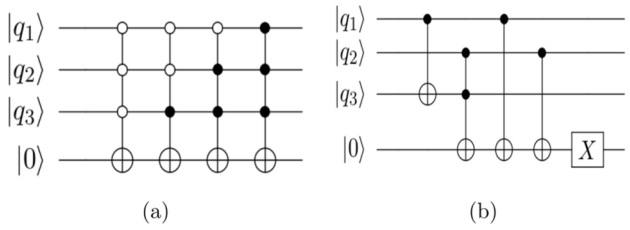


**Fig. 5**. (**a**) A circuit with 4 multi-controlled-NOT gates or $C^3X$, where each such gate is controlled on 3 qubits. (**b**) The same circuit implemented with 1 Toffoli gate.

$2N_1$ and $2N_2$ ancillae are required to implement $\mathrm{QRAM}_{poly,n_1}$ and $\mathrm{QRAM}_{poly,n_2}$, respectively. $\ell N_2$ qubits are required to copy the contents of the subset of memory locations selected by $\mathrm{QRAM}_{poly,n_1}$. $\ell N_2$ qubits are also required to parallelize the last $\ell N_2$ Toffolis and compute the parity.

**Comparison with previous works :** In the CSWAP architecture for qLUT[58] the authors combine a QROM and a specific QRAM. The QROM is implemented with a set of multi-controlled-NOT gates. The QRAM is implemented with a number of Fredkin or controlled-SWAP unitaries. Each controlled-SWAP can be implemented with a Toffoli and CNOT. The contents of the selected memory location is always obtained on some specified qubits. This qLUT has a T-count of $O(\sqrt{N})$, T-depth $O(\sqrt{N})$ and number of qubits $O(\sqrt{N})$.

If $N_1 = N_2 = \sqrt{N}$ then using any of the existing implementations of Toffoli, our $\mathrm{qLUT}_{poly}$ has T-depth $O(\log_2 \log_2 N)$, which is a double exponential improvement over the previous work. The T-count and number of qubits is asymptotically same. Thus from a fault-tolerant perspective, assuming the cost metric in Eq. (20), our design is expected to perform better.

Our CNOT cost is primarily dominated by the step where we copy a subset of memory locations selected by the first QRAM. We require $\ell N_1 N_2 = \ell N$ CNOTs at this stage. But again this is a group of $N_1$ multi-target CNOTs, where each has $N_2$ target. So in surface code implementation the execution time is equivalent to the time of execution of $N_1$ logical CNOTs. This cost is the same as that in[58].

## Toffoli-count optimization of quantum circuits

The polynomial encoding can be used to optimize the number of Toffolis required to implement groups of multi-controlled-X gates or mixed polarity multiple control Toffolis (MPMCTs)[29]. For example, we want to implement a circuit that flips a qubit to $|1\rangle$ for a subset, $\mathcal{S} \subseteq \{0,1\}^n$, of $n$-bit strings. These types of circuits also represent a kind of QROM. These can be used to select a subset of addresses and implement certain operations on those locations. We optimize the Toffoli-count of such circuits using the following procedure, which we call **TOFFOLI-OPT-POLY**.

1. Compute the encoding polynomial of each bit-string $\vec{b} \in \mathcal{S}$. This can be done conveniently using Lemma 2.
2. Compute the following sum of the encoding polynomials.

$$p(x_1, \ldots, x_n) = \bigoplus_{\vec{b} \in \mathcal{S}} p_{\vec{b}}(x_1, \ldots, x_n)$$

In this case coefficients of same monomials are added and reduced modulo 2.

3. A product of linear polynomials can be implemented with a Toffoli. We remember that a linear polynomial is the sum of monomials of weight at most 1. Such a polynomial can be implemented with CNOTs and X gates. Arrange the terms in $p(x_1, \ldots, x_n)$ such that the number of products of linear polynomials is optimized.
4. For each product we implement its factors in separate qubits, using CNOT and X gates. Using Toffoli we multiply these factors to implement the product. Then using CNOTs we add such product terms in order to implement $p(x_1, \ldots, x_n)$.

For illustration, consider the circuit shown in Fig. 5a, that has 3 qubits $q_1, q_2, q_3$ and another qubit initialized to $|0\rangle$. It flips the last qubit to $|1\rangle$ whenever the state of the first 3 qubits is $|000\rangle$, $|001\rangle$, $|011\rangle$ or $|111\rangle$. The encoding polynomials $p_{000}(x_1, x_2, x_3)$, $p_{001}(x_1, x_2, x_3)$, $p_{011}(x_1, x_2, x_3)$ and $p_{111}(x_1, x_2, x_3)$ have been calculated in Table 2. Then,

$$\begin{aligned} p(x_1, x_2, x_3) =& p_{000}(x_1, x_2, x_3) \oplus p_{001}(x_1, x_2, x_3) \oplus p_{011}(x_1, x_2, x_3) \oplus p_{111}(x_1, x_2, x_3) \\ =& 1 + x_1 + x_2 + x_1x_2 + x_2x_3 = 1 + x_1 + x_2 + x_2(x_1 + x_3). \end{aligned}$$

The 3 qubits $q_1, q_2, q_3$ are assigned variables $x_1, x_2, x_3$, respectively. With a CNOT controlled on $q_1$ and having target on $q_3$, we compute $x_1 \oplus x_3$. Then using a Toffoli controlled on $|q_2\rangle$ and $|q_3\rangle$, that store $x_2$ and $x_1 \oplus x_3$, respectively, we compute the product $x_2(x_1 + x_3)$. The rest of the variables can be added using CNOTs and X. The optimized circuit with 1 Toffoli gate has been shown in Fig. 5b.

Here we observe that a sequence of multi-controlled-X gates represents a Boolean function which is a sum of product terms. We can find its ESOP (Exclusive Sum-of-Products) expression using tools like EXORCISM[76–78]. Then factoring this expression we can implement the Boolean expression. We can also use the algorithm in[79] which first computes the ESOP, and then breaks the expression into common cofactors, which are reversibly synthesized. For example, in[29] the authors mentioned that the circuit in Fig. 5a can be implemented with 2 Toffolis. This is more than the Toffoli-count we get. We can also implement each multi-controlled-NOT-X gate using the decomposition given in[80]. Each $C^n X$ i.e. an $n$-qubit-controlled-X gate can be implemented with $n-1$ Toffolis and an additional $n-1$ ancillae. But this will give more Toffoli-count than our implementation in Fig. 5b.

## Discussions and conclusion

In this paper we develop a new design for quantum random access memory, using a polynomial encoding of the bit strings specifying the address of memory locations. We implement a Clifford+T circuit for our $\mathrm{QRAM}_{poly}$ and show that $\mathrm{QRAM}_{poly}$ has T-count $O(N-\log N-1)$, T-depth $O(\log\log N)$ and uses $O(N)$ logical qubits. Thus with our design of $\mathrm{QRAM}_{poly}$ we achieve an exponential improvement in T-depth, while reducing T-count and keeping the number of logical qubits requirement the same with respect to the previous state-of-the-art bucket brigade architecture[20,29]. We illustrate that when encoded with the surface code[64,65], in order to perform one memory read/write operation, $\mathrm{QRAM}_{poly}$ takes less time and uses much less number of physical qubits. Using two such $\mathrm{QRAM}_{poly}$ we implement a quantum look-up table ($\mathrm{qLUT}_{poly}$) that has T-count $O(\sqrt{N})$, T-depth $O(\log\log N)$ and uses $O(\sqrt{N})$ logical qubits. With our quantum look-up-table circuit $\mathrm{qLUT}_{poly}$ we achieve (Table 1) a double exponential improvement in T-depth over the previous state-of-the-art CSWAP architecture for qLUT[58], while the T-count and qubit-count are asymptotically same.

In our designs reduction in non-Clifford gate count comes at the cost of an increase in the CNOT gate count. The latter is a Clifford gate and in most error correction schemes the cost of implementing a Clifford is much less than the cost of implementing a non-Clifford. Thus, in our illustration we obtained better performance compared to previous QRAM designs. But CNOT, being a multi-qubit gate is more error prone than single-qubit gates like T. Even for connectivity constrained architectures (especially of the NISQ era) implementing a multi-qubit gate becomes more costly because it often needs a number of intermediate CNOT or SWAP gates, thus increasing the total gate count[81].

The problem of studying the noise-resilience of QRAM is an active research problem[27,28], especially for the pre-fault-tolerant regime. And we believe it is beyond the scope of this current work because this paper exclusively focuses on performance improvements in the fault-tolerant regime. Often metrics and design considerations in these two regimes differ and hence they are studied separately. A detail analysis of the noise-resilience of $\mathrm{QRAM}_{poly}$ is left for future work. It will also be interesting to study the mapping overhead of these circuits in different architectural layouts like 2D grid, as done in[36,59]. We expect to find trade-offs between the CNOT count and error rate or sparsity of the underlying graph. We can also aim at developing different hybrid designs with these new and existing circuits, so that we can take advantage of the various designs in different scenarios.

Using the polynomial encoding, we develop a method (TOFFOLI-OPT-POLY) to optimize the Toffoli-count of quantum circuits, especially those using multi-controlled-NOT gates. Since such circuits represent a sum-of-product (SOP) form of Boolean function, so these encodings can also have potential application in optimizing Boolean ESOP expressions, similar to the algorithms in[76–79]. Some of these classical algorithms have inspired methods for reversible quantum logic synthesis[73], which in turn have been an integral part of the design of quantum oracles for important algorithms like Grover's search. Thus these encoding polynomials may be used for reversible quantum logic synthesis. In the future we aim to investigate this avenue and the application of the polynomial encodings towards the design of algorithm-specific oracles and application-specific QRAMs, as in[40,49,72–74].

## Data availability

All relevant data are included in this manuscript.

Received: 19 November 2024; Accepted: 20 March 2025
Published online: 31 March 2025

## References

1. Grover, Lov K. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth Annual ACM Symposium on Theory of Computing*, pages 212–219, (1996).
2. van Apeldoorn, Joran, Gilyén, András, Gribling, Sander & de Wolf, Ronald. Convex optimization using quantum oracles. *Quantum* **4**, 220 (2020).
3. Reiher, Markus, Wiebe, Nathan, Svore, Krysta M., Wecker, Dave & Troyer, Matthias. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences* **114**(29), 7555–7560 (2017).
4. Babbush, Ryan et al. Low-depth quantum simulation of materials. *Physical Review X* **8**(1), 011044 (2018).
5. Bauer, Bela, Bravyi, Sergey, Motta, Mario & Chan, Garnet Kin-Lic. Quantum algorithms for quantum chemistry and quantum materials science. *Chemical Reviews* **120**(22), 12685–12717 (2020).
6. Cao, Yudong et al. Quantum chemistry in the age of quantum computing. *Chemical Reviews* **119**(19), 10856–10915 (2019).
7. Rubin, Nicholas C. et al. Fault-tolerant quantum simulation of materials using Bloch orbitals. *PRX Quantum* **4**(4), 040303 (2023).
8. Wilson Rosa de Oliveira. Quantum RAM based neural netoworks. *In ESANN* **9**, 331–336 (2009).
9. Biamonte, Jacob et al. Quantum machine learning. *Nature* **549**(7671), 195–202 (2017).
10. Harrow, Aram W., Hassidim, Avinatan & Lloyd, Seth. Quantum algorithm for linear systems of equations. *Physical Review Letters* **103**(15), 150502 (2009).
11. Huang, Hsin-Yuan., Kueng, Richard, Torlai, Giacomo, Albert, Victor V. & Preskill, John. Provably efficient machine learning for quantum many-body problems. *Science* **377**(6613), eabk3333 (2022).

12. Ciliberto, Carlo et al. Quantum machine learning: a classical perspective. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **474**(2209), 20170551 (2018).
13. Adcock, J. C. et al. *S Morley-Short* (AB Price, and S Stanisic. Advances in quantum machine learning. Quantum, S Pallister, 2015).
14. Bang, Jeongho, Dutta, Arijit, Lee, Seung-Woo. & Kim, Jaewan. Optimal usage of quantum random access memory in quantum machine learning. *Physical Review A* **99**(1), 012326 (2019).
15. Shor, Peter W. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE, (1994).
16. Kuperberg, Greg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *8th Conference on the Theory of Quantum Computation, Communication and Cryptography*, page 20, (2013).
17. Oh, Seunghyeok, Choi, Jaeho, Kim, Jong-Kook & Kim, Joongheon. Quantum convolutional neural network for resource-efficient image classification: A quantum random access memory (QRAM) approach. In *2021 International Conference on Information Networking (ICOIN)*, pages 50–52. IEEE, (2021).
18. Aaronson, Scott. Read the fine print. *Nature Physics* **11**(4), 291–293 (2015).
19. von Burg, Vera et al. Quantum computing enhanced computational catalysis. *Physical Review Research* **3**(3), 033055 (2021).
20. Giovannetti, Vittorio, Lloyd, Seth & Maccone, Lorenzo. Architectures for a quantum random access memory. *Physical Review A - Atomic, Molecular, and Optical Physics* **78**(5), 052310 (2008).
21. Konig, Robert, Maurer, Ueli & Renner, Renato. On the power of quantum memory. *IEEE Transactions on Information Theory* **51**(7), 2391–2401 (2005).
22. Blencowe, Miles. Quantum RAM. *Nature* **468**(7320), 44–45 (2010).
23. Liu, Chenxu, Wang, Meng, Stein, Samuel A, Ding, Yufei & Li, Ang. Quantum memory: A missing piece in quantum computing units. arXiv preprint arXiv:2309.14432, (2023).
24. Phalak, Koustubh, Chatterjee, Avimita & Ghosh, Swaroop. Quantum random access memory for dummies. *Sensors* **23**(17), 7462 (2023).
25. Kerenidis, Iordanis & Prakash, Anupam. Quantum recommendation systems. In *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, (2017).
26. Giovannetti, Vittorio, Lloyd, Seth & Maccone, Lorenzo. Quantum random access memory. *Physical Review Letters* **100**(16), 160501 (2008).
27. Arunachalam, Srinivasan, Gheorghiu, Vlad, Jochym-O'Connor, Tomas, Mosca, Michele & Srinivasan, Priyaa Varshinee. On the robustness of bucket brigade quantum RAM. *New Journal of Physics* **17**(12), 123010 (2015).
28. Hann, Connor T., Gideon, Lee, Girvin, S. M. & Jiang, Liang. Resilience of quantum random access memory to generic noise. *PRX Quantum* **2**(2), 020311 (2021).
29. Di Matteo, Olivia, Gheorghiu, Vlad & Mosca, Michele. Fault-tolerant resource estimation of quantum random-access memories. *IEEE Transactions on Quantum Engineering* **1**, 1–13 (2020).
30. Hong, Fang-Yu., Xiang, Yang, Zhu, Zhi-Yan., Jiang, Li.-zhen & Liang-neng, Wu. Robust quantum random access memory. *Physical Review A - Atomic, Molecular, and Optical Physics* **86**(1), 010306 (2012).
31. Moiseev, E. S. & Moiseev, S. A. Time-bin quantum RAM. *Journal of Modern Optics* **63**(20), 2081–2092 (2016).
32. Hann, Connor T. et al. Hardware-efficient quantum random access memory with hybrid quantum acoustic systems. *Physical Review Letters* **123**(25), 250501 (2019).
33. Chen, Kevin C., Dai, Wenhan, Errando-Herranz, Carlos, Lloyd, Seth & Englund, Dirk. Scalable and high-fidelity quantum random access memory in spin-photon networks. *PRX Quantum* **2**(3), 030319 (2021).
34. Pla, Jarryd. Chirping toward a quantum RAM. *Physics* **15**, 168 (2022).
35. Weiss, D. K., Puri, Shruti & Girvin, S. M. Quantum random access memory architectures using 3D superconducting cavities. *PRX Quantum* **5**(2), 020312 (2024).
36. Xu, Shifan, Hann, Connor T., Foxman, Ben, Girvin, Steven M & Ding, Yongshan. Systems architecture for quantum random access memory. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 526–538, (2023).
37. Amy, Matthew, Di Matteo, Olivia, Gheorghiu, Vlad, Mosca, Michele, Parent, Alex & Schanck, John. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In *International Conference on Selected Areas in Cryptography*, pages 317–337. Springer, (2016).
38. Soeken, Mathias, Roetteler, Martin, Wiebe, Nathan & De Micheli, Giovanni. LUT-based hierarchical reversible logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **38**(9), 1675–1688 (2018).
39. Jaques, Samuel & Schanck, John M. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39*, pages 32–61. Springer, (2019).
40. Park, Daniel K., Petruccione, Francesco & Rhee, June-Koo Kevin. Circuit-based quantum random access memory for classical data. *Scientific Reports* **9**(1), 3949 (2019).
41. De Veras, Tiago ML., De Araujo, Ismael CS., Park, Daniel K. & Da Silva, Adenilton J. Circuit-based quantum random access memory for classical data with continuous amplitudes. *IEEE Transactions on Computers* **70**(12), 2125–2135 (2020).
42. Asaka, Ryo, Sakai, Kazumitsu & Yahagi, Ryoko. Quantum random access memory via quantum walk. *Quantum Science and Technology* **6**(3), 035004 (2021).
43. Zidan, Mohammed, Abdel-Aty, Abdel-Haleem., Khalil, Ashraf, Abdel-Aty, Mahmoud & Eleuch, Hichem. A novel efficient quantum random access memory. *IEEE Access* **9**, 151775–151780 (2021).
44. Dangwal, Siddharth, Sharma, Ritvik & Bhowmik, Debanjan. Fast-QTrain: an algorithm for fast training of variational classifiers. *Quantum Information Processing* **21**(5), 189 (2022).
45. Asaka, Ryo, Sakai, Kazumitsu & Yahagi, Ryoko. Two-level quantum walkers on directed graphs. ii. application to quantum random access memory. *Physical Review A* **107**(2), 022416 (2023).
46. Bugalho, Luís. et al. Resource-efficient simulation of noisy quantum circuits and application to network-enabled QRAM optimization. *npj Quantum Information* **9**(1), 105 (2023).
47. Clarino, David, Asada, Naoya & Yamashita, Shigeru. Optimizing LUT-based quantum circuit synthesis using relative phase Boolean operations. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE, (2023).
48. Chen, Zhao-Yun, Xue, Cheng, Wang, Yun-Jie, Sun, Tai-Ping, Liu, Huan-Yu, Zhuang, Xi-Ning, Dou, Meng-Han, Zou, Tian-Rui, Fang, Yuan, Wu, Yu-Chun, et al. Efficient and error-resilient data access protocols for a limited-sized quantum random access memory. arXiv preprint arXiv:2303.05207, (2023).
49. Phalak, Koustubh, Li, Junde & Ghosh, Swaroop. Trainable PQC-based QRAM for quantum storage. *IEEE Access* **11**, 51892–51899 (2023).
50. Liu, Junyu, Hann, Connor T. & Jiang, Liang. Data centers with quantum random access memory and quantum networks. *Physical Review A* **108**(3), 032610 (2023).
51. Liu, Junyu & Jiang, Liang Quantum data center: Perspectives. *IEEE Network*, (2024).
52. Duan, Bojia & Hsieh, Chang-Yu. Compact and classically preprocessed data-loading quantum circuit as a quantum random access memory. *Physical Review A* **110**(1), 012616 (2024).
53. Hunt, Ethan. Phase RAM: Phase estimation's application in QRAM. In *Proceedings of the 2024 ACM Southeast Conference*, pages 205–210, (2024).

54. Mukhopadhyay, Priyanka, Stetina, Torin F. & Wiebe, Nathan. Quantum simulation of the first-quantized Pauli-Fierz Hamiltonian. *PRX Quantum* **5**(1), 010345 (2024).
55. Berry, Dominic W., Gidney, Craig, Motta, Mario, McClean, Jarrod R. & Babbush, Ryan. Qubitization of arbitrary basis quantum chemistry leveraging sparsity and low rank factorization. *Quantum* **3**, 208 (2019).
56. Häner, Thomas, Kliuchnikov, Vadym, Roetteler, Martin & Soeken, Mathias. Space-time optimized table lookup. arXiv preprint arXiv:2211.01133, (2022).
57. Krishnakumar, Rajiv, Soeken, Mathias, Roetteler, Martin & Zeng, William. A Q# implementation of a quantum lookup table for quantum arithmetic functions. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*, pages 75–82. IEEE, (2022).
58. Low, Guang Hao, Kliuchnikov, Vadym & Schaeffer, Luke. Trading T gates for dirty qubits in state preparation and unitary synthesis. *Quantum* **8**, 1375 (2024).
59. Zhu, Shuchen, Sundaram, Aarthi & Low, Guang Hao. Unified architecture for a quantum lookup table. arXiv preprint arXiv:2406.18030, (2024).
60. Mosca, Michele & Mukhopadhyay, Priyanka. A polynomial time and space heuristic algorithm for T-count. *Quantum Science and Technology* **7**(1), 015003 (2021).
61. Mukhopadhyay, Priyanka. CS-count-optimal quantum circuits for arbitrary multi-qubit unitaries. *Scientific Reports* **14**(1), 13916 (2024).
62. Mukhopadhyay, Priyanka. Synthesizing Toffoli-optimal quantum circuits for arbitrary multi-qubit unitaries. arXiv preprint arXiv:2401.08950, (2024).
63. Mukhopadhyay, Priyanka. Synthesis of V-count-optimal quantum circuits for multiqubit unitaries. *Physical Review A* **109**(5), 052619 (2024).
64. Fowler, Austin G., Mariantoni, Matteo, Martinis, John M. & Cleland, Andrew N. Surface codes: Towards practical large-scale quantum computation. *Physical Review A - Atomic, Molecular, and Optical Physics* **86**(3), 032324 (2012).
65. Fowler, Austin G. Time-optimal quantum computation. arXiv preprint arXiv:1210.4626, (2012).
66. Häner, Thomas & Soeken, Mathias. Lowering the T-depth of quantum circuits by reducing the multiplicative depth of logic networks. arXiv preprint arXiv:2006.03845, (2020).
67. Gheorghiu, Vlad, Mosca, Michele & Mukhopadhyay, Priyanka. A (quasi-) polynomial time heuristic algorithm for synthesizing T-depth optimal circuits. *npj Quantum Information* **8**(1), 110 (2022).
68. Jones, Cody. Low-overhead constructions for the fault-tolerant Toffoli gate. *Physical Review A - Atomic, Molecular, and Optical Physics* **87**(2), 022328 (2013).
69. Gidney, Craig. Halving the cost of quantum addition. *Quantum* **2**, 74 (2018).
70. Selinger, Peter. Quantum circuits of T-depth one. *Physical Review A - Atomic, Molecular, and Optical Physics* **87**(4), 042302 (2013).
71. Paler, Alexandru, Oumarou, Oumarou & Basmadjian, Robert. Parallelizing the queries in a bucket-brigade quantum random access memory. *Physical Review A* **102**(3), 032608 (2020).
72. Niu, Murphy Yuezhen et al. Entangling quantum generative adversarial networks. *Physical Review Letters* **128**(22), 220505 (2022).
73. Seidel, Raphael et al. Automatic generation of Grover quantum oracles for arbitrary data structures. *Quantum Science and Technology* **8**(2), 025003 (2023).
74. Nagy, Ákos & Zhang, Cindy. Novel oracle constructions for quantum random access memory. arXiv preprint arXiv:2405.20225, (2024).
75. Gheorghiu, Vlad, Mosca, Michele & Mukhopadhyay, Priyanka. T-count and T-depth of any multi-qubit unitary. *npj Quantum Information* **8**(1), 141 (2022).
76. Song, Ning & Perkowski, Marek A. Exorcism-mv-2: minimization of exclusive sum of products expressions for multiple-valued input incompletely specified functions. In *[1993] Proceedings of the Twenty-Third International Symposium on Multiple-Valued Logic*, pages 132–137. IEEE, (1993).
77. Song, Ning & Perkowski, Marek A. Minimization of exclusive sum-of-products expressions for multiple-valued input, incompletely specified functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **15**(4), 385–395 (1996).
78. Mishchenko, Alan & Perkowski, Marek. Fast heuristic minimization of exclusive-sums-of-products. (2001).
79. Shafaei, Alireza, Saeedi, Mehdi & Pedram, Massoud. Reversible logic synthesis of k-input, m-output lookup tables. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1235–1240, IEEE, (2013).
80. He, Yong, Ming-Xing Luo, E., Zhang, Hong-Ke Wang. & Wang, Xiao-Feng. Decompositions of n-qubit Toffoli gates with linear circuit complexity. *International Journal of Theoretical Physics* **56**, 2350–2361 (2017).
81. Gheorghiu, Vlad, Huang, Jiaxin, Li, Sarah Meng, Mosca, Michele & Mukhopadhyay, Priyanka. Reducing the CNOT count for Clifford+ T circuits on NISQ architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **42**(6), 1873–1884 (2022).


## Acknowledgements

The author thanks Nathan Wiebe for helpful discussions. The author also thanks the anonymous reviewers whose helpful comments have helped us improve our manuscript significantly. The author acknowledges funding from the NSERC discovery program.

## Author contributions

The ideas, implementations and preparation of the manuscript was done by P.Mukhopadhyay.

## Declarations

## Competing interests

The author declares no competing interests.

## Additional information

**Correspondence** and requests for materials should be addressed to P.M.

**Reprints and permissions information** is available at www.nature.com/reprints.

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.