Quantum circuit design for universal distribution using a superposition of classical automata

Aritra Sarkar, Zaid Al-Ars, Koen Bertels
Department of Quantum & Computer Engineering
Delft University of Technology, The Netherlands

April, 2021

Abstract

In this research, we present a quantum circuit design and implementation for a parallel universal linear bounded automata. This circuit is able to accelerate the inference of algorithmic structures in data for discovering causal generative models. The computation model is practically restricted in time and space resources. A classical exhaustive enumeration of all possible programs on the automata is shown for a couple of example cases. The precise quantum circuit design that allows executing a superposition of programs, along with a superposition of inputs as in the standard quantum Turing machine formulation, is presented. This is the first time, a superposition of classical automata is implemented on the circuit model of quantum computation, having the corresponding mechanistic parts of a classical Turing machine. The superposition of programs allows our model to be used for experimenting with the space of program-output behaviors in algorithmic information theory. Our implementations on OpenQL and Qiskit quantum programming language is copy-left and is publicly available on GitHub.

Keywords: quantum Turing machines, quantum circuits, models of computation, linear bounded automata, causality
1 Introduction

The phenomenal success of data-driven approaches like deep learning has ushered automation in many spheres of human society over the last decade. However, such black-box optimization often fail to provide insights on the mechanism underlying the set of observations about the physical process under study. This limitation in explainability are increasingly becoming crucial with automation in sectors like healthcare. In contrast, symbolic approaches have been successfully used to model, study and understand the causal relationships in natural phenomena and datasets. For example, in genomics, it finds applications [1] in metabiology, phylogenetic tree analysis and protein-protein interaction mapping. A better understanding of the algorithmic structures in DNA sequences would greatly advance personalized medication, drug discovery and synthetic biology.

Defining an algorithmic process via symbolic manipulations requires a computation model. The Turing machine model of computation is the cornerstone of theoretical computer science. This simple mechanistic automata has the expressive power to capture any algorithmic process. Thus, it can be used for generic modeling and hypothesis comparison across various scientific disciplines.

The set of transformations a computation model can undergo and the resulting space of outputs is central to understanding the causal structure of the physical phenomena we intend to model. Except for the trivial cases, this remains intractable on classical computers since the space of all possible transformations grows exponentially with the number of states and symbols of the automata. Thus, in practice they are approximated by restricting the resources available to the computational model like time/cycles and space/memory. Even with such restrictions, relatively simple automata have been explored using supercomputing clusters [2]. These results have found various applications in genomics, psychology, network science, image processing, etc.

The quantum computation model provides distinctive advantages for specific algorithms using the laws of quantum mechanics. Recently, a generic (gate-based) circuit model of quantum Turing machine was proposed [3] based on cellular automata. In this work, we propose an alternative model from a mechanistic perspective, with a detailed circuit implementation with realistic assumptions on run-time and qubit resources for the tape memory. We do away with the homogeneous local structure resulting in execution of many inactive unitary transforms, thereby improving the total number of executed operations. Our model intuitively allows encoding a superposition of classical programs and evaluating their evolution after a predetermined number of cycles.

In this research, we present the exact scalable circuit using standard quantum gates required to simulate a superposition of this resource-bounded stored-program automata. With the recent thrust in the realization of quantum computing hardware, this is a promising direction with multifaceted application that can extend the applications of approximating algorithmic metrics by enumerating automata configurations. The availability of better quantum processors would allow this algorithm to be readily ported on a quantum accelerator with a quantum computing stack [4]. We are motivated by applications in soft-computing, specifically estimating lower semi-computable metrics like algorithmic probability and Kolmogorov complexity, with application in genomics, artificial life and artificial intelligence.

The necessary background of classical automata models is presented in §2. Thereafter, the quantum automata models and their circuit implementations are reviewed in §3. In §4 we propose our computation model and its key features that distinguish it from other approaches. An exhaustive enumeration of a few small examples of the model is presented. The quantum implementation of the QPULBA model is presented in §5. The resources and circuit designs of each functional block of the quantum algorithm are worked out for one of the example cases of 2 state and 2 symbol automata. In §6 we present our results of the implementation of two cases of the QPULBA on two quantum programming platforms, OpenQL and Qiskit. §7 concludes the paper.
2 Classical automata models

In this section, we briefly present the Turing machine model of computation and the variants of the model relevant for further discussions in the paper. This is presented from the perspective of the Chomsky hierarchy of classical automata and languages.

2.1 Turing machine model

The Turing machine (TM) is the canonical model of computation invented by Alan Turing for proving the uncomputability of the Entscheidungsproblem. A TM manipulates symbols on an infinite memory strip of tape divided into discrete cells according to a table of transition rules. These user-specified transition rules can be expressed as a finite state machine (FSM) which can be in one of a finite number of states at any given time and can transition between states in response to external inputs. The Turing machine, as shown in figure 1, positions its head over a cell and reads the symbol there. As per the read symbol and its present state in the table of instructions, the machine (i) writes a symbol (e.g. a character from a finite alphabet) in the cell, then (ii) either moves the tape one cell left or right, and (iii) either proceeds to a subsequent instruction or halts the computation.

![Figure 1: Computational model of a Turing machine](image)

Each model of computation defines a set of inputs that are accepted by that automata. For the Turing machine, this subset (of all possible strings over the alphabet of the language) that can be enumerated (outputs the string) is called recursively enumerable. Recursively enumerable languages (also called, Turing recognizable/acceptable, semi/partially decidable) forms the type-0 in the Chomsky hierarchy of formal languages, as discussed in §2.3.

2.2 Variants of the TM model

While Turing machines can express arbitrary mechanical computations, their minimalist design makes them unsuitable for algorithm design for computation in practice. Various modifications of the TM results in equivalent computation power, as captured by the Church-Turing thesis. Here, equivalent refers to being within polynomial translation overhead in time or memory resources. Thus, in practice they might compute faster, use less memory, or have smaller instruction set, but they cannot compute more mathematical functions. These modified models of computation, like lambda calculus, Post machine, cyclic-tag system, offers different perspective in development of computer hardware and software. Extending the TM tape to multiple dimensions is also equivalent to a 1 dimensional tape (or, only 1-way infinite tape).

In this section, we discuss some variants of the Turing machine model, as listed in Table 1, that are relevant for further ideas developed in this paper. These variants can also be applied to the corresponding language instead of the automata. Turing completeness is the ability for a system of instructions to simulate a Turing machine. A Turing complete programming language is thus theoretically capable of expressing all tasks that can be accomplished by computers. Nearly all programming languages are Turing complete if the limitations of finite memory are ignored.

A universal Turing machine (UTM) simulates an arbitrary Turing machine on an arbitrary input. Every TM can be assigned a number, called the machine’s description number, similar to Gödel numbers, encoding the FSM as a list of transitions. The existence of this direct correspondence between natural numbers and TM implies that the set of all Turing machines (or programs of a fixed size) is denumerable. A UTM reads the description of the machine to be simulated as well as the input to that machine from its own tape, as...
2.3 Chomsky hierarchy of automata and languages

For real implementations of an automata an infinite tape is not possible. Thus computational models restrict the tape features in various ways. These result in a reduced level in the Chomsky hierarchy of formal languages and the corresponding automata model that accepts strings from the respective grammar.

- **Type-0**: Recursively enumerable language; Turing machine (TM)
- **Type-1**: Context-sensitive language; Linear bounded automata (LBA)
- **Type-2**: Context-free language; Pushdown automaton (PDA)
- **Type-3**: Regular languages; Finite state machine (FSM)

Each higher level (the highest being type-0) can always simulate the lower level. Restrictions in memory makes the other levels computationally less capable than a TM in terms of the language, (i.e. the set of string patterns called the grammar) it can recognize. For example, an FSM cannot determine if an input string has the structure $a^i b^j$, like $aabb$, $aaaabb$. At type-0, universality is reached, thus, everything that is computable can be mapped to an algorithmic process on the TM. This includes quantum computation as well, as it can be simulated on a classical TM (albeit using worst-case exponential resources).

A linear bounded automata (LBA) has a finite memory, extending the same modifications as for the TM, as listed as the fourth modification to TM models in Table 1 forming the type-1 in the Chomsky hierarchy. Most real-world computers can be best modeled as the Parallel Linear Bounded Random Access Stored Program (PLBRASP) automata. This allows multiple concurrent processors with a shared random access finite sized memory. The program as well as the data are accessed from the memory in the von Neumann architecture.

In this work, we will consider the Parallel Universal Linear Bounded Automata (PULBA) model which is the sequential memory access variant of the PLBRASP. Thus, we consider multiple stored programs running in parallel while sharing a common limited work memory and output interface.

2.4 Cellular automata model

Often for physical system, a cellular automata model is preferred as an alternative to Turing machines, specifically when there is a homogeneous local interaction, e.g. in geographic spacial planning and ecological systems. Cellular automata is a discrete model of computation consisting of a regular grid of cells in any finite dimension (or type of tessellation). Each cell at any step in time is in one of a finite number of states,
Table 1: Variants of the Turing machine model for deterministic, non-deterministic and quantum automata classes. Our implementation of QPULBA (in yellow) captures the computing capabilities of 27 (in blue) out of 51 automata models (of type 3, 2, 1) that is realistically implementable on physical hardware.

| Automata class | Features |
|----------------|----------|
|                | Type     | Deterministic | Non-deterministic | Quantum | Stored program | Random memory access | Parallel | Infinite memory | Movable tape head | Memory based |
| 1              |          | FSM          | NFSM            | QFSM     | N             | N                 | N         | N               | N               | N           |
| 2              |          | PDA          | NPDA            | QPDA     | N             | N                 | N         | N               | N               | Y           |
| 3              |          | UPDA         | NUPDA           | QUPDA    | Y             | N                 | N         | N               | N               | Y           |
| 4              |          | PDRAA        | NPDRAA          | QPDRAA   | N             | Y                 | N         | N               | N               | Y           |
| 5              |          | PDRASP       | NPDRAA          | QPDRAA   | Y             | Y                 | N         | N               | N               | Y           |
| 6              |          | PPDA         | NPPDA           | QPPDA    | N             | N                 | Y         | N               | N               | Y           |
| 7              |          | PUPDA        | NPUPDA          | QPUPDA   | Y             | N                 | N         | Y               | N               | Y           |
| 8              |          | PDRAA        | NPDRAA          | QPDRAA   | Y             | Y                 | N         | N               | N               | Y           |
| 9              |          | PPDRASP      | NPPDRASP        | QPPDRASP | Y             | Y                 | Y         | N               | N               | Y           |
| 10             |          | LBA          | NLBA            | QLBA     | N             | N                 | N         | N               | Y               | Y           |
| 11             |          | ULBA         | NULBA           | QULBA    | Y             | N                 | N         | N               | Y               | Y           |
| 12             |          | LBRAM        | NLBRAM          | QLBRAM   | N             | Y                 | N         | N               | N               | Y           |
| 13             |          | LBRASP       | NLBRASP         | QLBRASP  | Y             | Y                 | N         | Y               | Y               | Y           |
| 14             |          | PLBA         | NPLBA           | QPLBA    | N             | N                 | Y         | Y               | Y               | Y           |
| 15             |          | PULBA        | NPULBA          | QPULBA   | Y             | N                 | Y         | N               | Y               | Y           |
| 16             |          | PLBRAM       | NPLBRAM         | QPLBRAM  | N             | Y                 | Y         | Y               | Y               | Y           |
| 17             |          | PLBRASP      | NPLBRASP        | QPLBRASP | Y             | Y                 | Y         | N               | Y               | Y           |
| 18             |          | TM           | NTM             | QTM      | N             | N                 | N         | Y               | Y               | Y           |
| 19             |          | UTM          | NUTM            | QUTM     | N             | N                 | Y         | Y               | Y               | Y           |
| 20             |          | RAM          | NRAM            | QRAM     | N             | Y                 | N         | Y               | Y               | Y           |
| 21             |          | RASP         | NRASP           | QRASP    | Y             | Y                 | N         | Y               | Y               | Y           |
| 22             |          | PTM          | NPTM            | QPTM     | N             | N                 | Y         | Y               | Y               | Y           |
| 23             |          | PUTM         | NPUTM           | QPUTM    | Y             | N                 | Y         | Y               | Y               | Y           |
| 24             |          | PRAM         | NPRAM           | QPRAM    | N             | Y                 | Y         | Y               | Y               | Y           |
| 25             |          | PRASP        | NPRASP          | QPRASP   | Y             | Y                 | Y         | Y               | Y               | Y           |
(such as on and off). For each cell, a set of cells called its neighborhood is defined relative to the specified cell. The next step in time is created according to some fixed rule (typically same over the entire space) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. The whole grid typically updates simultaneously. The model with a 1-dimensional tape over 2 symbols and a neighborhood size of 3 (1 cell in either size along with the current cell), is called elementary cellular automata (ECA). These were studied extensively revealing interesting features like the Rule 110 (one of the 256 possible rules) capable of universal computation [7].

Solid-state implementation [8] of cellular automata involves expressing the update rule in terms of classical logic gates (e.g. AND, OR, NOT, NAND) as shown in figure 3 for ECA Rule 110. For each cell, the cell and its two immediate neighbors are read, their values entered into the transition function, and the cell updated with the new value for the subsequent generation. The transition function can be represented as a truth table, a logic function or a logic circuit. The result from this function is then written to the corresponding cell in a duplicate array. In this example, transition functions are being calculated from $C_L$ and written sequentially to $C_R$. Once a generation is complete, the original array ($C_L$ here) is cleared, and the results of rule applications to cells in $C_R$ are written to $C_L$, like a ping-pong buffer.

A cellular automata can simulate a TM by storing a activation bit indicating the position of the tape head and the current state for each cell. The transition function of the TM is encoded as the update rule depending only on the current cell. Thus, it is like PTM with identical FSM, with only one of them active at each cycle. This model is wasteful in terms of logic gate operations as only one of the cells is active yet all the other logic gates need to be executed none the less.
Quantum computing uses the laws of quantum mechanics to a computational advantage. These includes reversible unitary transformation of quantum states utilizing superposition and entanglement. The algorithm design involves harnessing constructive interference between solution state’s complex amplitudes and destructive interference between non-solutions before an irreversible projective measurement is performed on a chosen basis. A good algorithm results in a high probability of reading out the classical state of an acceptable solution to the application.

A quantum mechanical model of Turing machines was described by Paul Benioff using Hamiltonian models [9, 10]. A computationally equivalent model using quantum gates (inspired by classical Boolean logic gates) in a quantum circuit was proposed by David Deutsch [11, 12, 13]. Due to the more intuitive nature of the circuit model it has become the standard for quantum algorithm development. A generalized quantum Turing machine (GQTM) [14], which contains QTM as a special case and includes non-unitary dynamics (irreversible transition functions) as well, allow the representation of quantum measurements without classical outcomes [15]. From a quantum computer architecture perspective, QTM was compared [16] to quantum random-access machine (QRAM) and quantum random-access stored-program (QRASP) model, proving their equivalence in bounded-error polynomial-time formulations using the Solvay-Kitaev theorem. The QRASP considers that a single program is stored in classical registers, and thus treated as classical data, while the work memory can be in a superposition. A QRASP model of a quantum computer can encode a program as quantum data, consequently generalizing the QRASP model to parallel quantum random-access stored-program machines (PQRASP) allowing a superposition of programs [17].

Here we review the circuit implementations of quantum finite automata (QFA) [18, 19], quantum equivalent automata models [20], and more extensively QTM [3] which inspires our circuit architecture of a quantum parallel universal linear bounded automata (QPULBA). Like its classical counterpart, a quantum Turing machine (QTM) is an abstract model capturing the power of quantum mechanical process for computation. Any quantum algorithm can be expressed formally as a particular QTM.

### 3.1 Quantum Turing machines

A classical TM is extended to the complex vector space for a QTM. For a three-tape QTM (one tape holding the input, a second tape holding intermediate calculation results, and a third tape holding output):

- The set of states is replaced by a Hilbert space.
- The tape alphabet symbols are replaced by a Hilbert space (usually a different Hilbert space than the set of states).
- The blank symbol corresponds to the zero-vector.
- The input and output symbols are usually taken as a discrete set, as in the classical system; thus, neither the input nor output to a quantum machine need be a quantum system itself.
- The transition function is a generalization of a transition monoid. It is a collection of unitary matrices that are automorphisms of the Hilbert space.
- The initial state may be either a mixed state or a pure state.
- The set of final or accepting states is a subspace of the Hilbert space.

It is important to realize that this is merely a sketch than a formal definition of a quantum Turing machine. Some important details like how often a measurement is performed are not explicitly defined. This is circumvented by formal proofs showing equivalence with an oblivious QTM, whose running time is a function of the size of the input (independent of the structure of the input). However, how to practically translate that to a priori knowledge of the number of steps the computation needs to be executed before collapsing the superposition still needs further exploration [21, 22].

### 3.2 Cellular automata inspired QTM circuit

QTM was developed in the 1990s as a formal model to represent quantum computation. Recently, [3] revived this direction of research by presenting quantum circuit formulations and discussing applications of simulating quantum automata.

The approach taken in [3] resembles a (quantum) cellular automata in its homogeneous circuit template and local unitary operations. The no-cloning principle prevents a direct translation of the sequentially updating classical cellular automata architecture to quantum circuits. This is because, when the state is sequentially updated, we lose the neighbor information to update the consecutive cell. The circuit architecture of this QTM is inspired by the solid-state implementation of the computational equivalent model to a TM of a 1-dimensional elementary cellular automata. The local simultaneous update is achieved by maintaining a local one-hot marker denoting the presence/absence of the tape head at the location activates a specific
(or superposition of) computation path in the QTM. It defines a fixed unitary $G$ which encodes the rule, or the transition function and interleaved in a particular way as shown in figure 4. For the quantum version, the inputs are qubits, so they can be in a superposition of states, allowing simultaneous evolution of various inputs (initial Turing tapes).

By this approach, $t \geq n$ steps of a QTM on an input of length $n$ can be simulated by a uniformly generated family of quantum circuits linear in $t$. However, not all transition functions describe valid quantum Turing machines as it is imperative that the global evolution is unitary on the Hilbert space. Any QTM that exhibits a local causal behavior can be mapped to the transition function $G$ proposed in [3]. Since the model is universal, there exists a transition function that will in effect read a stored program and the input from the tape as qubits and perform the computation entirely by local operations. However, it is not immediately clear how to construct such a unitary for an arbitrary transition function. Moreover, modeling the stored-program model on this local interaction architecture would incur a high cost in the number of operations to affect the tape, as at each step a TM head moves only by one step.
4 Proposed computation model

In this section, we propose the formulation of our computation model that we translate to the quantum circuit in §5. Our motivation is to complement the research as presented in [3] from a different perspective. The major considerations to make the quantum circuit practically implementable and tractable in resources is discussed.

4.1 A mechanistic perspective

The intricacy in applying the cellular automata based QTM to arbitrary algorithms inspires our research to look at more intuitive model with structural similarity to a TM. We do away with the local transition function applied in parallel based on activation qubits. Our quantum circuit provides a mechanistic model of a Linear Bounded Automata, while allowing both the inputs (tape memory) and the program (transition function) to be in superposition. Thus, in effect it provides a scalable quantum circuit implementation for the QPULBA model of computation.

The superposition of programs is the key feature allowing the framework to be applied for estimating algorithmic features [23], like the algorithmic probability and algorithmic complexity. Note that, this parallelism we propose is different from the superposition of inputs as considered by standard QTM models [3], as the parallelism we propose allows both the input data as well as the transition function to be in a superposition. As an analogy, this can be thought of as the distinction between a quantum adder (which can add a superposition of two inputs), and a quantum calculator (which can apply all possible binary operations like add, subtract, multiply, divide, power, etc. on the two inputs). Each classical binary operation need not access the inputs from the tape in a coordinated fashion, e.g. multiplication can proceed from right to left while division can proceed from left to right. Once the quantum algorithm is executed, the output evolves to a superposition considering all possible programs (with the given states and symbols) on all possible inputs - commonly referred to as the Solomonoff universal prior probability distribution in algorithmic information theory.

4.2 Bounds on runtime

For any pragmatic (quantum) automata implementation, besides the memory, the runtime also needs to be restricted to a predetermined cycle count. This is imperative as it is not possible to estimate the halting time of a Turing machine in advance. Not all applications however allow a time restricted formulation. In our research, we are interested in these specific types of algorithms which can be aborted externally (with or without inspecting the current progress). The premature output yields an acceptable approximate solution with a bounded error based on the computation time. They come under the purview of soft-computing and are ubiquitous in optimization, machine learning and evolutionary algorithms.

Here we consider runtime restriction in the context of a linear-bounded automata. There are 2 variables in a LBA specification, apart from the alphabet and states. These are the length of computation (time) and the length of the tape (space) resources. The latter is the dependent variable due to the region of influence (time cone). It is possible for a code to run for infinite time on a limited tape, but it is not possible to cover an infinite tape in a finite time. Thus, time-restricted automata further restrict the set of strings that are reachable by the computation process. There is no definitive metric to compare the power of runtime restricted automata models without invoking algorithmic information metrics like the speed prior or logical depth. Most landmark research [24] on the universality of TMs has assumed infinite tape length, adding a few points on the pareto curve of universality for the number of symbols and states. The runtime of our model is thus application and resource driven, based on the convergence of the model for the specific phenomena, and the coherence limits of the quantum processor.

Additionally, we do not consider halting states specifically in our model. However, it is not too difficult to consider the subset of programs with halting states and are by default realized when a state loops on itself. Recent research [25] has shown that a non-halting TM is statistically correlated in rank with LBA. Though the Chomsky hierarchy level of a non-halting LBA has not been explored, we presume that it would also be correlated to LBA rather than to PDA or FSM. As discussed later, restricting the automata based on the number of cycles is an unavoidable technicality in the quantum version as we need to collapse the superposition.

4.3 Corresponding classical model

Before we present the quantum model of QPULBA, in this section, we define the corresponding classical model where neither the program nor the input can be in a superposition. Thus, it corresponds to a restricted
runtime ULBA, where the program is accepted from outside the system (e.g. stored in a program memory) and the input tape is finite in length.

In our computation model, we will restrict a \( m \) states, \( n \) symbols, 1 dimension tape Turing machine by limiting the maximum time steps \( t \) before a forced halt is imposed. This automatically bounds the causal cone on the tape to \([-t, +t]\) from the initial position of the tape head, resulting in a LBA.

The tape is initialized to the blank character as this does not reduce the computational power of the TM. This can be thought of as, the initial part of the program prepares the input on the tape and then computes on it. The tape length, like the memory size of CPU, is an application specific hyperparameter chosen such that it is enough for accommodating the intermediate work-memory scratchpad and the final output. The range of values for the tape length is \( c \leq (2t + 1) \).

The generic model of a restricted ULBA is represented by this 10-tuple.

\[
T = (Q, \Gamma, b, \Sigma, \delta, q_0, F, t, d, c)
\]

- \( Q \) is a finite, non-empty set of states.
- \( \Gamma \) is a finite, non-empty set of symbols allowed on the tape, called the tape alphabet.
- \( b \in \Gamma \) is the blank symbol.
- \( \Sigma \subseteq \Gamma \setminus \{b\} \) is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents.
- \( \delta \) is a partial function called the transition function. It defines the next state, tape movement and write symbol based on the current state and the read symbol.
- \( q_0 \in Q \) is the initial state.
- \( F \subseteq Q \) is the set of final states or accepting states.
- \( t \) is the number of steps the machine is executed before a forced halt is imposed.
- \( d \) is the dimension of the tape. It also specifies if the tape is circular by a \( \circ \) symbol for each dimension.
- \( c \) is the length of the tape in each dimension.

For our experimental implementation example in this paper, we chose \( c = t \) and a circular tape. This helps us evaluate output diversity considering tape direction equivalence under left-right substitution. This shorter tape however comes as the cost of not being able to map to computational path dependent application where left-right substitution is not always equivalent (e.g. robotic movement). Our computation model of \( m = |Q| \) states, \( n = |\Gamma| \) symbols, \( d = 1 \) dimension (circular) tape restricted Turing machine, can be formally represented as:

\[
T = \{(Q_0, Q_1, \ldots, Q_{m-1}), \{s_0, s_1, \ldots, s_{n-1}\}, s_0, \{\}, \delta, Q_0, \{\}, 1^c, c\}
\]

Note that \( \Sigma \) is empty, thus the tape is always initialized to the blank character \( b = s_0 \). The set of accepting states \( F \) is also empty to prevent the machine from halting before \( t \) steps. This includes machines that have defined halting states, by modifying the transition function to remain in the same state and write the same symbol that is read (in effect simulating a no-operation) once these states are reached.

Thus, the transition table exhaustively lists a transition for each combination of \( (Q - F) = Q \) and \( \Gamma \).

\[
\delta : Q \times \Gamma \to Q \times \Gamma \times \{0, 1\}
\]

where, 0 is left shift, 1 is right shift for the 1 dimensional tape. Each entry of the transition table requires \( \log_2(m) + \log_2(n) + d \) bits and there are a total of \( m \times n \) entries. Thus, the number of bits required to specify a single transition function is:

\[
q_3 = (m \times n) \times (\log_2(m) + \log_2(n) + d)
\]

The number of different machines (programs) that can be represented using \( q_3 \) bits is represented by:

\[
P = 2^{q_3}
\]

### 4.4 Enumeration of example cases

Here, some examples of this computation model are enumerated. These enumerations are intended to be executed in parallel thereby presenting the PULBA variant of the ULBA model. The cases are labeled as per the number of states, symbols and tape dimension (Case \( m-n-d \)). We start with the smallest natural number 1 and explore larger values of symbols and states (however, we will only consider 1 dimensional Turing tape).

For these experiments, the number of iterations is set equal to the size of the program, i.e. \( t = q_3 \). This allows us to compare the final tape and program to infer which programs can self-replicate, which motivates our research [23].
4.4.1 Case 1-1-1

For this case, the number of states $m = 1$ with the state set $Q : \{Q_0\}$. The alphabet is $\Gamma : \{0, 1\}$, thus, $n = 1$ (the unary alphabet). This gives the values $q_3 = 1 \ast 1 \ast (0 + 0 + 1) = 4$ and $P = 2^4 = 16$.

The machine is run for $t = q_3 = 4$ iteration. The initial tape is of length $c = t = 1$. To distinguish the blank character from a written 0, we will use $0$. Thus, the initial tape is: $0000$.

The outputs for each program (description number) are listed in Table 2. $R_s/W_s$ refers to the read/write symbols and $M_{l/r}$ refers to the tape movement of left/right.

Table 2: Exhaustive enumeration of the programs of Case 1-1-1 circular tape ULBA for length 1 cycle 1

| P#  | $Q_0R_0$ | Final tape |
|-----|--------|------------|
| 0   | $Q_0M_1W_0$ | 0          |
| 1   | $Q_0M_1W_0$ | 0          |

4.4.2 Case 2-1-1

For this case, the number of states $m = 2$ with the state set $Q : \{Q_0, Q_1\}$. The alphabet is again $\Gamma : \{0, 1\}$, with, $n = 1$ (the unary alphabet). This gives the values $q_3 = 2 \ast 1 \ast (1 + 0 + 1) = 4$ and $P = 2^4 = 16$.

The machine is run for $t = q_3 = 4$ iteration. The initial tape is of length $c = t = 4$. The initial tape is: $0000$, where the underline denotes the initial position.

The outputs for each program (description number) are listed in Table 3. It is easy to see all $m$-1-d cases will result in tapes with the blank/zero characters. The 0000 string is the only string in this language.

Table 3: Exhaustive enumeration of the programs of Case 2-1-1 circular tape ULBA for length 4 cycle 4

| P#  | $Q_1R_0$ | $Q_0R_0$ | Final tape |
|-----|---------|----------|------------|
| 00  | $Q_0M_1W_0$ | $Q_0M_1W_0$ | 0000       |
| 01  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 02  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 03  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 04  | $Q_0M_1W_0$ | $Q_0M_2W_0$ | 0000       |
| 05  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 06  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 07  | $Q_0M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 08  | $Q_1M_1W_0$ | $Q_0M_1W_0$ | 0000       |
| 09  | $Q_1M_1W_0$ | $Q_0M_1W_0$ | 0000       |
| 10  | $Q_1M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 11  | $Q_1M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 12  | $Q_1M_1W_0$ | $Q_2M_1W_0$ | 0000       |
| 13  | $Q_1M_1W_0$ | $Q_0M_1W_0$ | 0000       |
| 14  | $Q_1M_1W_0$ | $Q_1M_1W_0$ | 0000       |
| 15  | $Q_1M_1W_0$ | $Q_1M_1W_0$ | 0000       |

4.4.3 Case 1-2-1

For this case, the number of states $m = 1$ with the state set $Q : \{Q_0\}$. The alphabet is $\Gamma : \{0, 1\}$, with, $n = 2$ (the binary alphabet). This gives the values $q_3 = 1 \ast 2 \ast (1 + 1 + 0) = 4$ and $P = 2^4 = 16$. The machine is run for $t = q_3 = 4$ iteration. The initial tape is of length $c = t = 4$. The initial tape is similar to the last case: $0000$.

The outputs for each program (description number) are listed in Table 4. If we run the experiment for more than 4 steps then we will see more variety in the output.

4.4.4 Case 2-4-1

Let us consider the case where the number of states $m = 2$ (with the states set $Q : \{Q_0, Q_1\}$) with the alphabet $\Gamma : \{A, C, G, T\}$, with, $n = 4$, inspired by the DNA alphabet. This gives the values $q_3 = 2 \ast 4 \ast (1 + 2 + 1) = 32$ and $P = 2^{32} = 4294967296$. It is clear that even for the simple case of the DNA
Table 4: Exhaustive enumeration of the programs of Case 1-2-1 circular tape ULBA for length 4 cycle 4

| P# | Q₀R₁ | Q₀R₀ | Final tape |
|----|------|------|------------|
| 00 | Q₀M₁W₀ | Q₀M₁W₀ | 0000       |
| 01 | Q₀M₁W₀ | Q₀M₁W₁ | 1111       |
| 02 | Q₀M₁W₀ | Q₀M₁W₀ | 0000       |
| 03 | Q₀M₁W₀ | Q₀M₁W₁ | 1111       |
| 04 | Q₀M₁W₁ | Q₀M₁W₀ | 0000       |
| 05 | Q₀M₁W₁ | Q₀M₁W₁ | 1111       |
| 06 | Q₀M₁W₁ | Q₀M₁W₀ | 0000       |
| 07 | Q₀M₁W₁ | Q₀M₁W₁ | 1111       |
| 08 | Q₀M₁W₀ | Q₀M₁W₀ | 0000       |
| 09 | Q₀M₁W₀ | Q₀M₁W₁ | 1111       |
| 10 | Q₀M₁W₀ | Q₀M₁W₀ | 0000       |
| 11 | Q₀M₁W₀ | Q₀M₁W₁ | 1111       |
| 12 | Q₀M₁W₁ | Q₀M₁W₀ | 0000       |
| 13 | Q₀M₁W₁ | Q₀M₁W₁ | 1111       |
| 14 | Q₀M₁W₁ | Q₀M₁W₀ | 0000       |
| 15 | Q₀M₁W₁ | Q₀M₁W₁ | 1111       |

An alphabet, an exhaustive search by enumeration is not possible on a classical algorithm (and maybe even on a quantum computer simulator running on classical hardware). This exponential growth in the number of cases shows the difficulty of classical enumeration of ULBA motivating quantum acceleration [26, 27] for bioinformatics applications.

4.4.5 Case 2-2-1

This case is both non-trivial as well as within the bounds of our current experimentation. The number of states \( m = 2 \) with the state set \( Q : \{Q₀, Q₁\} \). The alphabet is \( Γ : \{0, 1\} \), thus, \( n = 2 \) (the binary alphabet). This gives the values \( q_δ = 2^2 * (1 + 1) = 12 \) and \( P = 2^{12} = 4096 \) using the formula discussed before.

The machine is run for \( t = q_δ = 12 \) iteration. The initial tape of length \( c = t = 12 \) is: 000000000000

The program (description number) is encoded as:

\[
[QMW]^Q₁R₁[QMW]^Q₁R₀[QMW]^Q₀R₁[QMW]^Q₀R₀
\]

There are too many cases to enumerate by hand, so a Python script (the classical kernel we intend to accelerate) is written that emulates our restricted model of the linear bounded automata for all 4096 cases. The program can be found at the following link:

https://github.com/Advanced-Research-Centre/QuBio/blob/master/Project\_01/classical/

The tape output for all the 4096 machines is plotted in figure 5.

Figure 5: Tape output for all programs of Case 2-2-1 circular tape ULBA for length 12 cycle 12
5 QPULBA: quantum parallel universal linear bounded automata

In this section, we present the detailed design of the quantum circuit to implement the ULBA computation model of the previous section. This is a mechanistic model of a QPULBA having the corresponding parts of a classical ULBA as qubits. The acronym expansion of ‘quantum’, ‘parallel’, ‘universal’, ‘linear bounded’ translates respectively to the automata features of a superposition in inputs, a superposition of programs, a stored-program model and a memory restricted implementation. As highlighted in Table 1, QPULBA (in yellow) captures the computing capabilities of 27 (in blue) out of 51 automata models (of type 3, 2, 1) that is realistically implementable on physical hardware. The circuit in figure 6 requires some ancilla qubits which will be introduced later. The automata step needs to be repeated for the number of steps \( t \) we intend to execute the machine before measuring out the qubits.

![QPULBA step](image)

Figure 6: Blocks for the quantum circuit implementation of a QPULBA step

5.1 Qubit complexity

The qubit complexities of the design elements are discussed here. The generic formula is derived before applying to the specific case of the 2-2-1 QPULBA of § 4.4.5.

- **Alphabet**: Alphabet set cardinality \( n = |\Gamma| \) is the number of symbols in the alphabet. The number of bits/qubits required to represent a symbol, \( q_{\Gamma} = \lceil \log_2(n) \rceil \)
- **Head position**: The current head position is represented either as binary or one-hot encoding. The one-hot encoding is more expensive in the number of qubits, but better in terms of gates. The number of bits/qubits required for one-hot encoding \( c \) is the same as the number of cells \( c \). Since the simulation bottleneck is the number of qubits instead of the number of gates, we prefer the binary encoding. For binary encoding, \( q_{\text{head}} = \lceil \log_2(c) \rceil \)
- **Read head**: The read head temporarily stores the content of the current head position, requiring bits/qubits, \( q_{\text{read}} = q_{\Gamma} \)
- **Write head**: The write head temporarily stores the content to be written to the current head position, requiring bits/qubits, \( q_{\text{write}} = q_{\Gamma} \)
- **Turing tape**: The number of bits/qubits required for the restricted tape size of \( c \) is, \( q_{\text{tape}} = c \times q_{\Gamma} \)
- **Movement**: Specifying the movement of a \( d \) dimensional Turing tape requires, \( q_{\text{move}} = d \)
- **Current state**: The current state in binary encoding requires, \( q_{\text{state}} = \lceil \log_2(m) \rceil \). In a one-hot coded format, it would require \( m \) qubits.
- **Transition table**: The transition function is a unitary matrix that transforms the input and the current state to the output, next state and movement of the tape. Thus, for each combination of state and read character, we need to store the next state, write character and movement. The number of qubits required are, \( q_{\delta} = (m \times n) \times (q_{\text{state}} + q_{\text{write}} + q_{\text{move}}) \)
- **Computation history**: Since the quantum circuit is reversible, the computation history for \( (t-1) \) steps needs to be stored in ancilla qubits. The computation history is specified by the state and read...
symbol for each step, requiring, $q_{ch} = (t - 1) \ast (q_{state} + q_{read})$. However, it is possible to trade-off space (qubits) with time (operations) by uncomputing, as discussed in § 5.3.5. Thus, the qubit complexity of the implementation (assuming $q_a$ ancilla qubits) is:

$$\begin{align*}
q_{QPULBA} = & q_\delta + q_{state} + q_{move} + q_{head} + q_{read} + q_{write} + q_{tape} + q_{ch} + q_{a} \\
= & (m \ast n \ast \log(m) + \log(n) + d) + \log(m) + d + \log(c) + \log(n) + \log(n) + (c \ast \log(n)) \\
& + q_{ch} + q_{a}
\end{align*}$$

Considering the 2-2-1 QPULBA example, the values of $m = 2$, $n = 2$, $d = 1$, $c = 12$, $t = 12$ are substituted in the above equation (all logarithms are base-2 and rounded up to be nearest integer) to yield,

$$\begin{align*}
q_{QPULBA}^{221} = & (2 \ast 2 \ast (\log(2) + \log(2) + 1)) + \log(2) + 1 + \log(12) + \log(2) + \log(2) + (12 \ast \log(2)) \\
& + (11 \ast (\log(2) + \log(2))) + q_{a} \\
= & 12 + 1 + 1 + 4 + 1 + 1 + 12 + 22 + q_{a} \\
= & 54 + q_{a}
\end{align*}$$

Simulating in order of 50 qubits is near the quantum supremacy limits. However, the circuit is not always in full superposition thereby allowing smart simulation techniques in a quantum simulator and uncomputing away the computation history.

### 5.2 Initialize

The initialization circuit depends on the target application for this framework. For measuring the algorithmic probability or the universal distribution, all possible programs (represented by the transition table) need to be evolved in a superposition. All other qubits are kept at the ground or default state of $|0\rangle$. The circuit is shown in figure 7.

![Figure 7: Initialization quantum circuit for QPULBA 2-2-1](image)

### 5.3 QPULBA step

Each iteration of the QPULBA undergoes the following transforms:

1. Read: $\{q_{read}\} \leftarrow U_{read}(\{q_{head}, q_{tape}\})$
2. Transition evaluation: $\{q_{move}, q_{ch}, q_{write}\} \leftarrow U_{\delta}(\{q_{read}, q_{state}, q_{\delta}\})$
3. Write: $\{q_{tape}\} \leftarrow U_{write}(\{q_{head}, q_{write}\})$
4. Move: $\{q_{head}\} \leftarrow U_{move}(\{q_{head}, q_{move}\})$
5. Reset

This corresponds to one step of a classical UTM, with the distinction of the computation now evolving in a superposition of all possible classical automata. We will now discuss each of these QPULBA steps in detail.
5.3.1 Read tape

The quantum circuit implements a multiplexer with the tape as the input signals and the binary coded head position as the selector lines. The read head is the output. The read circuit for the QPULBA 2-2-1 is shown in figure 8.

Figure 8: Read tape quantum circuit for QPULBA 2-2-1
5.3.2 Transition table lookup

The transition table encoding is:

\[ [Q_t | R_t] \rightarrow [Q_{t+1} | M_{t\ell} | W_T] \]

Note that we use \( q_{\text{state}}^- \) instead of \( q_{\text{state}}^+ \) though we are storing the next state in the qubit. This is corrected by the reset circuit. The transition function circuit for the QPULBA 2-2-1 is shown in figure 9.

![Transition function quantum circuit for QPULBA 2-2-1](image)

Figure 9: Transition function quantum circuit for QPULBA 2-2-1
5.3.3 Write tape

The quantum circuit implements a de-multiplexer with the tape as the output signals and the head position as the selector lines. The write head is the input.

The write circuit for the QPULBA 2-2-1 is shown in figure 10. The qubit elements not involved are not shown.

![Write tape quantum circuit for QPULBA 2-2-1](image-url)
5.3.4 Move

There are many choices for implementing the move, e.g. a looped tape (overflow/underflow is ignored and trimmed), error flag is raised and halts, overflow/underflow is ignored, etc. Here, a looped tape is implemented.

The head is incremented/decremented using the move qubit as control.

\[
\begin{array}{c}
|q_{\text{head}}\rangle \\
\text{Inc} \\
|q_{\text{move}}\rangle \end{array}
\quad
\begin{array}{c}
\text{Dec} \\
X \\
X
\end{array}
\]

The increment/decrement circuit is a special case of the quantum full adder \cite{28} with the first register, a set to 1. For the QPULBA 2-2-1 case, the length of the circular tape is 12, thus, the increment and decrement needs to be modulo 12. For increment, when the \(q_{\text{head}}\) equals 11, it should increment to \((11 + 1) \mod 12 = 0\), while for decrement, \((0 - 1) \mod 12 = 11\). Thus, for these edge cases, we need to increment/decrement by 5 instead of 1. We set the \(a_2\) bit to change the effective value of \(a\) from \(1 = 0001_2\) to \(5 = 0101_2\) for the addition/subtraction. This operation is conditioned on the head value and move bit, and denoted as the overflow/underflow qubit \(|\text{ovfw}\rangle/|\text{udfw}\rangle\). The \(a_2\) bit is uncomputed based on the incremented value being 0 or the decremented value being 11.

The carry (\(C\)), sum (\(S\)) and reverse carry (\(C^*\)) blocks are defined as follows:

\[
\begin{array}{c}
.\text{sum} \\
cnot A0,S0 \\
cnot B0,S0
\end{array}
\quad
\begin{array}{c}
.\text{carry} \\
toffoli A0,B0,C1 \\
cnot A0,B0 \\
toffoli C0,B0,C1 \\
toffoli A0,B0
\end{array}
\quad
\begin{array}{c}
.\text{reverse_carry} \\
toffoli C0,B0,C1 \\
toffoli A0,B0,C1
\end{array}
\]
Increment

In this design, $c_3c_2c_1 = 000$ are 3 ancilla ($c_0$ is not required), $a_0 = q_{\text{move}}, a_3a_2a_1 = 000$, $b_3b_2b_1b_0 = q_{\text{head}}q_{\text{head}}q_{\text{head}}q_{\text{head}}$ and $b_4$ is ignored. The circuit in figure 11 shows the 4-bit modulo-12 quantum increment circuit using the quantum adder blocks.

![Figure 11: Modulo-12 quantum adder for implementing move tape head for QPULBA 2-2-1](image)

The circuit can be simplified considering some of the control qubits are always in the 0 state, so those CNOT/Toffoli gates can be ignored. The optimized increment circuit for the QPULBA 2-2-1 is shown in figure 12.

![Figure 12: Modulo-12 increment quantum circuit for QPULBA 2-2-1](image)
Decrement
In this design, $c_3a_2c_1 = 000$ are 3 ancilla ($c_0$ is not required), $a_0 = q_{move}$, $a_3a_2a_1 = 000$, $b_3b_2b_1b_0 = q_{head}q_{head}q_{head}q_{head}$ and $b_4$ is ignored. The circuit in figure 13 shows the 4-bit modulo-12 quantum decrement circuit using the reverse order of blocks as the standard quantum adder.

The circuit can be simplified considering some of the control qubits are always in the 0 state, so those CNOT/Toffoli gates can be ignored. The optimized increment circuit for the QPULBA 2-2-1 is shown in figure 14.
Quantum logic is universal and can implement any classical Boolean logic function using only the Toffoli (CCNOT) or Fredkin (CSWAP) gate. However, it is not always possible to uncompute all ancilla qubits. Specifically, if the function to be implemented is irreversible, e.g. AND, extra qubits are needed to construct the reversible quantum circuit that preserves the unitary property. The general strategy to compile a classical function to quantum logic is shown in figure 15.

Other research has focused on the quantum circuit generation of the constrained subset of reversible automata. However, for the QPULBA case, we intend to evolve a superposition of all possible classical functions/programs $C_f$ that can be represented by the description number encoding. These functions include both reversible as well as irreversible functions, thus, we cannot uncompute away the computation history of the state transition.

Both the state and the read together preserve the evolution history. Thus, we need ancilla qubits in each step of the computation that would hold the transition history for the QPULBA. This limits the number of steps of the QPULBA we can implement or simulate. Besides the state and read, the write and move qubits need to be reset in each cycle. This is implemented by calling the FSM transition function once again with the previous state and the read.
6 Implementation and simulation results

In this section, we present the circuit implementation of QPULBA. This was implemented on 2 different programming platforms, OpenQL [29] developed at the Delft University of Technology and IBM’s Qiskit [30]. Our copy-left AGPLv3 licensed implementation can be found on: https://github.com/Advanced-Research-Centre/QPULBA

We implemented 2 cases of QPULBA, with 1 and 2 states: the full circuit and cycle simulation of QPULBA 1-2-1, and a limited simulation of the units for QPULBA 2-2-1, as presented below.

6.1 QPULBA 1-2-1

Our implementation is scalable to any \(m\)-state \(n\)-symbol QPULBA. The entire circuit for the 1-state 2-symbol case requires much less qubits, thus we were able to simulate it classically. Note that there is no need to store the state anymore thereby reducing the qubit complexity greatly.

Number of 1-state 2-symbol 1-dimension QPULBA: 16

FSM : \([0, 1, 2, 3]\)
STATE : \([]\)
MOVE : \([4]\)
HEAD : \([5, 6]\)
READ : \([7]\)
WRITE : \([8]\)
TAPE : \([9, 10, 11, 12]\)
ANCILLA : \([13, 14, 15]\)

The full circuit was simulated for 4 cycles as discussed in §4(d) 4.4.3. The final state vector obtained after 4 cycles is shown in figure 16. The FSM qubits encoding the description/program number (in green) and the output on the tape (in red) bit strings match with the classical enumeration in Table 4. Thus, if we measure only the tape in the standard computational basis, we will obtain an equal statistical distribution of the 0000 and 1111 states.

Figure 16: Test result for the QPULBA 1-2-1 showing the FSM description/program number (green) and output tape (red)

6.2 QPULBA 2-2-1

The entire circuit for the 2-state 2-symbol case can be compiled from the parts described in §5 This was implemented in a scalable manner on OpenQL and Qiskit.
6.2.1 Full circuit compilation

The qubit allocation for the full circuit, considering 1 cycle is:

Number of 2-state 2-symbol 1-dimension QPULBA: 4096

FSM : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
STATE : [12, 13]
MOVE : [14]
HEAD : [15, 16, 17, 18]
READ : [19]
WRITE : [20]
TAPE : [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
ANCILLA : [33, 34, 35]

For each further cycles, we need 2 qubits to store the computation history.

The exact gate complexity depends on the considered primitive. We use Hadamard, Pauli-X, CNOT, Toffoli and SWAP as our gate set. Multi-qubit controlled-NOT gates are decomposed using the borrowed-ancilla strategy outlined in [31]. One cycle of the QPULBA uses 627 gates: 476 Toffoli, 126 Pauli-X, 12 CNOT, 12 Hadamard and 1 SWAP gate. The Qiskit circuit drawing and generated OpenQASM can be found on the repository.

6.2.2 Unit tests

While we were able to compile the full circuit, the large number of qubits limits classically simulating the circuit on our available hardware. The exponential simulation complexity of quantum algorithms on classical hardware in terms of memory resource is indeed the driving factor for research on physical implementation of quantum accelerators. To complement our design, we developed unit tests for each part of the mechanistic perspective.

![Figure 17: Test circuit for the QPULBA 2-2-1 move block](image)

We successfully tested each of the 6 parts of the QPULBA, i.e. initialize, read, FSM, write, move and reset. The unit test simulations are tractable as each part concerns only a subset of the qubits. The inputs are put into an unequal superposition of values using random rotations about the Y-axis (that maintains the amplitude in the real domain). This allows us to individually inspect each basis state changes in contrast to an equal superposition using Hadamard gates. For quantum acceleration of classical algorithms, the ZX-plane of the Bloch sphere is enough to take advantage of the destructive interference of quantum superpositions over classical probabilistic computing, thereby reducing the cost of classical simulation using complex representations of the amplitude.
To track the output changes, each qubit of the target register is entangled with a test register using CNOT gates. Thus, the old value and the new value of the basis state’s bit string can be inspected similar to an associative memory.

Here, as an example, we show the unit test circuit for the move step, which requires 8 qubits for the circuit and 4 addition qubits for testing. The circuit is shown in figure 17. The initial part before the first barrier is the test configuration and the second part is of the move circuit.

For the move circuit, the binary string of the qubits associated with the head state (in red) increments by 1 if the move qubit (in blue) is 1 and decrements by 1 otherwise. This is verified by the internal state vector output of the simulation as shown in figure 18.

```
| State Vector |
|--------------|
| (+0.03251+0.00000j) | 0000000011>
| (+0.10148+0.00000j) | 00000001110>
| (+0.09940+0.00000j) | 00010000000>
| (+0.03184+0.00000j) | 00010000101>
| (+0.10876+0.00000j) | 00100000010>
| (+0.03484+0.00000j) | 00100000111>
| (+0.10652+0.00000j) | 00110000100>
| (+0.03412+0.00000j) | 00110001001>
| (+0.03820+0.00000j) | 01000000110>
| (+0.01224+0.00000j) | 01000001011>
| (+0.03741+0.00000j) | 01110001000>
| (+0.01198+0.00000j) | 01110000110>
| (+0.04094+0.00000j) | 01110000111>
| (+0.04010+0.00000j) | 01110001100>
| (+0.01284+0.00000j) | 01110001101>
| (+0.42239+0.00000j) | 10000001110>
| (+0.13530+0.00000j) | 10000001011>
| (+0.41369+0.00000j) | 10010000000>
| (+0.13252+0.00000j) | 10010000101>
| (+0.45268+0.00000j) | 10100000100>
| (+0.14500+0.00000j) | 10100001011>
| (+0.44336+0.00000j) | 10110000100>
| (+0.14202+0.00000j) | 11110000110>
| (+0.15899+0.00000j) | 11110000110>
| (+0.05093+0.00000j) | 11110000110>
| (+0.15571+0.00000j) | 11110000110>
| (+0.04988+0.00000j) | 11110000110>
| (+0.17039+0.00000j) | 11110000110>
| (+0.05458+0.00000j) | 11110000110>
| (+0.05346+0.00000j) | 11110000110>
| (+0.16688+0.00000j) | 11110000110>
```

Figure 18: Test result for the QPULBA 2-2-1 move block showing the move (blue), head (green), ancilla (yellow) and test (green) qubits
7 Conclusion

The mechanistic model of computation as exhibited by a Turing machine defines an algorithm as an initial input to final output transformation on the tape memory by a program defined as a finite state machine. The set of transformations a computation model can undergo and the resulting space of outputs is central to understanding the causal structure of a physical phenomena for scientific modeling and hypothesis testing. While it has many applications, except for the trivial cases, this remain intractable on classical computers. This is because the space of all possible transformations grows exponentially with the number of states and symbols of the automata.

In this research, we explore the distinctive advantages for classical automata simulation offered by the alternate paradigm of quantum computation. We complement the recently proposed \cite{3} circuit design of a quantum Turing machine from a mechanistic perspective with realistic assumptions on runtime and qubit resources. In our design, we follow the computation model of a quantum parallel universal linear bounded automata.

We present the exact scalable circuit using standard quantum gates required to simulate a superposition of programs of this automata, thereby obtaining the distribution of their evolution after a predetermined number of cycles. This algorithm can be readily ported on a quantum accelerator stack \cite{4} with any sufficiently advanced gate-model quantum computing hardware, in terms of qubits, connection topology and error rates. We present our results of the implementation of two cases of the automata on two quantum programming platforms, OpenQL and Qiskit. We simulated and verified 4 cycles of the 1-state 2-symbol quantum parallel universal linear bounded automata with 16 qubits and compiled and unit tested the 2-state variant that requires 36 qubits.

Quantum automata has promising advantages in extending the applications of approximating algorithmic metrics by enumerating automata configurations \cite{32}, as will be explored in our future research. Applications in soft-computing, specifically accelerating the estimating of metrics like algorithmic probability \cite{23} and algorithmic complexity, are essential primitives in the fields of genomics, artificial life and artificial intelligence.
References

[1] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Estimating algorithmic information using quantum computing for genomics applications. Applied Sciences, 11(6):2696, 2021.

[2] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. Calculating kolmogorov complexity from the output frequency distributions of small turing machines. PloS one, 9(5), 2014.

[3] Abel Molina and John Watrous. Revisiting the simulation of quantum turing machines by quantum circuits. Proceedings of the Royal Society A, 475(2226):20180767, 2019.

[4] K Bertels, A Sarkar, T Hubregtsen, M Serrao, AA Mouedenne, A Yadav, A Krol, and I Ashraf. Quantum computer architecture: Towards full-stack quantum accelerators. In 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2020.

[5] David B Searls. The language of genes. Nature, 420(6912):211–217, 2002.

[6] Marc D Hauser and Jeffrey Watumull. The universal generative faculty: The source of our expressive power in language, mathematics, morality, and music. Journal of Neurolinguistics, 43:78–94, 2017.

[7] Stephen Wolfram. A new kind of science, volume 5. Wolfram media Champaign, IL, 2002.

[8] Jack C Chaplin, Natalio Krasnogor, and Noah A Russell. Photochromic molecular implementations of universal computation. Biosystems, 126:12–26, 2014.

[9] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. Journal of statistical physics, 22(5):563–591, 1980.

[10] Paul Benioff. Quantum mechanical hamiltonian models of turing machines. Journal of Statistical Physics, 29(3):515–546, 1982.

[11] David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences, 400(1818):97–117, 1985.

[12] David Elieser Deutsch. Quantum computational networks. Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences, 425(1868):73–90, 1989.

[13] A Chi-Chih Yao. Quantum circuit complexity. In Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science, pages 352–361. IEEE, 1993.

[14] Satoshi Iriyama, Masanori Ohya, and Igor Volovich. Generalized quantum turing machine and its application to the sat chaos algorithm. In Quantum Information and Computing, pages 204–225. World Scientific, 2006.

[15] Simon Perdrix and Philippe Jorrand. Classically controlled quantum computation. Mathematical Structures in Computer Science, 16(4):601–620, 2006.

[16] Qisheng Wang and Mingsheng Ying. Quantum random access stored-program machines. arXiv preprint arXiv:2003.03514, 2020.

[17] Mingsheng Ying, Li Zhou, and Yangjia Li. Reasoning about parallel quantum programs. arXiv preprint arXiv:1810.11334, 2018.

[18] Michael R Dunlavey. Simulation of finite state machines in a quantum computer. arXiv preprint quant-ph/9807026, 1998.

[19] AC Cem Say and Abuzeb Yakaryilmaz. Quantum finite automata: A modern introduction. In Computing with New Resources, pages 208–222. Springer, 2014.

[20] Edison Tsai and Marek Perkowski. A quantum algorithm for automata encoding. FactaUniversitatis, Series: Electronics andEnergetics, 33(2):169–215, 2020.

[21] Noah Linden and Sandu Popescu. The halting problem for quantum computers. arXiv preprint quant-ph/9806054, 1998.

[22] Stefano Guerrini, Simone Martini, and Andrea Masini. Quantum turing machines computations and measurements. arXiv preprint arXiv:1703.07748, 2017.

[23] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Estimating algorithmic information using quantum computing for genomics applications. Preprints, 2021.

[24] Turlough Neary and Damien Woods. Small weakly universal turing machines. In International Symposium on Fundamentals of Computation Theory, pages 262–273. Springer, 2009.
[25] Hector Zenil, Liliana Badillo, Santiago Hernández-Orozco, and Francisco Hernández-Quiroz. Coding-theorem like behaviour and emergence of the universal distribution from resource-bounded algorithmic probability. *International Journal of Parallel, Emergent and Distributed Systems*, 34(2):161–180, 2019.

[26] Aritra Sarkar, Zaid Al-Ars, Carmen G Almudever, and Koen Bertels. An algorithm for dna read alignment on quantum accelerators. *arXiv preprint arXiv:1909.05563*, 2019.

[27] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Quaser–quantum accelerated de novo dna sequence reconstruction. *arXiv preprint arXiv:2004.05078*, 2020.

[28] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147, 1996.

[29] Nader Khammassi, Imran Ashraf, J v Someren, Razvan Nane, AM Krol, M Adriaan Rol, L Lao, Koen Bertels, and Carmen G Almudever. Openql: A portable quantum programming framework for quantum accelerators. *arXiv preprint arXiv:2005.13283*, 2020.

[30] Ryan LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[31] Craig Gidney. Constructing large controlled nots. Available at [http://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Not.html](http://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Not.html), 2018.

[32] Hector Zenil. Experimental algorithmic information theory — complexity and randomness. Available at [http://www.mathrix.org/experimentalAIT/](http://www.mathrix.org/experimentalAIT/), 2020.