On the Universality of Memcomputing Machines

Yan Ru Pei, Fabio L. Traversa, and Massimiliano Di Ventra

***Abstract*—Universal memcomputing machines (UMMs) [IEEE Trans. Neural Netw. Learn. Syst. 26, 2702 (2015)] represent a novel computational model in which memory (time non-locality) accomplishes both tasks of storing and processing of information. UMMs have been shown to be Turing-complete, namely they can simulate any Turing machine. In this paper, using set theory and cardinality arguments, we compare them with liquid-state machines (or "reservoir computing") and quantum machines ("quantum computing"). We show that UMMs can simulate both types of machines, hence they are both "liquid-" or "reservoir-complete" and "quantum-complete". Of course, these statements pertain only to the type of problems these machines can solve, and not to the amount of resources required for such simulations. Nonetheless, the method presented here provides a general framework in which to describe the relation between UMMs and any other type of computational model.**

***Index Terms*—memory, elements with memory, Memcomputing, Turing Machine, Reservoir Computing, Quantum Computing.**

I. Introduction

Memcomputing stands for "computing *in* and *with* memory" [1]. It is a novel computing paradigm whereby memory (time non-locality) is employed to perform both tasks: storing and processing of information on the same physical location. This paradigm is substantially different than the one implemented in our modern-day computers [2]. In these machines there is a separation of tasks between a memory (storage) unit and one that performs the processing of information. Modern-day computers are the closest physical approximation possible to the well-known (mathematical) Turing paradigm of computation that maps a finite string of symbols into a finite string of symbols in discrete time [3].

The memcomputing paradigm has been formalized by Traversa and Di Ventra in Ref. [4]. In that paper it was shown that universal memcomputing machines (UMMs) can be defined as digital (so-called *digital memcomputing machines* (DMMs) [5]) or analog [6], with the digital ones offering an easy path to scalable (in terms of size) machines.

UMMs have several features that make them a powerful computing model, most notably *intrinsic parallelism*, *information overhead*, and *functional polymorphism* [4]. The first feature means that they can operate *collectively* on all (or portions of) the data at once, in a cooperative fashion. This is reminiscent of neural networks, and indeed neural networks can be viewed as special cases of UMMs. However, neural networks do not have "information overhead". This feature is related to the *topology* (or architecture) of the network of memory units (*memprocessors*). It means the machine has access, at any given time, to more information (precisely originating from the topology of the network) than what is available were the memprocessors not connected to each other. Of course, this information overhead is not necessarily stored by the machine (the *stored* information is the Shannon one) [4]. Nevertheless, with appropriate topologies, hence with appropriate information overhead, UMMs can solve complex problems very efficiently [4]. Finally, functional polymorphism means that the machine is able to compute different functions without modifying the topology of the machine network, by simply applying the appropriate input signals [4].

In Ref. [4] it was shown that UMMs are Turing-*complete*, meaning that UMMs can simulate *any* Turing machine. Note that Turing-completeness means only that all problems a Turing machine can solve, can also be solved by a UMM. It does *not* imply anything about the resources (in terms of time, space and energy) required to solve those problems.

The reverse, namely that all problems solved by UMMs are also solved by Turing machines (namely *equivalence* between the two models of computation) has not been proved yet. Nevertheless, a practical realization of *digital* memcomputing machines, using electronic circuits with memory [5], has been shown to be efficiently simulated with our modern-day computers (see, e.g., [7]). In other words, the ordinary differential equations representing DMMs and the problems they are meant to solve can be efficiently simulated on a classical computer.

In recent years, other computational models, each addressing different types of problems, have attracted considerable interest in the scientific community. On the one hand, *quantum computing*, namely computing using features, like tunneling or entanglement, that pertain only to quantum phenomena, has become a leading candidate to solve specific problems, such as prime factorization [8], annealing [9], or even simulations of quantum Hamiltonians [10]. This paradigm has matured from a simple proposal [11] to full-fledged devices for possible commercial use [12]. However, its scalability faces considerable practical challenges, and its range of applicability is very limited compared to even classical Turing machines.

Another type of computing model pertains to a seemingly different domain, the one of spiking (recurrent) neural networks, in which spatio-temporal patterns of the network can be used to, e.g., learn features after some training [13]. Although somewhat different realizations of this type of networks have been suggested, for the purposes of this paper, here we will focus only on the general concept of "reservoir-computing" [14], and in particular on its "liquid-state machine" (LSM) realization [15], rather than the "echo-state network" one [16]. The results presented in this paper carry over to this other type of realization as well. Therefore, we will use the term "reservoir-computing" as analogous to "liquid-state machine"

YRP and MD are with the Department of Physics, University of California-San Diego, 9500 Gilman Drive, La Jolla, California 92093-0319, USA, (e-mail: yrpei@ucsd.edu, diventra@physics.ucsd.edu). FLT is with MemComputing Inc., San Diego, CA 92130 USA (e-mail: ftraversa@memcpu.com).

and will not differentiate among the different realizations of the former.

Our goal for this paper is to study the relation between these seemingly different computational models. In particular, we will show that UMMs encompass both quantum machines and liquid-state machines, in the sense that, on a theoretical level, they can simulate *any* quantum computer or *any* liquid-state machine. Again, this does not say anything about the *resources* needed to simulate such machines, only that such a mapping exists. In other words, we prove here that UMMs are not only Turing-complete, but also "liquid-complete" (or "reservoir-complete") and "quantum-complete".

In order to prove these statements, we will use set theory and cardinality arguments to show that the LSM and quantum computers can be mapped to subsets of the UMM. This methodology is general. Therefore, we expect it to be applicable to any other type of computational model, other than the ones we consider in this work.

Our paper is organized as follows. In Sec. II we briefly review the mathematical definition of the machines we consider in this work, starting with UMMs. In Sec. III we introduce the basic ingredients of set theory that are needed to prove the completeness statements that we will need later. In Sec. IV we show how to define a general computing model in terms of set theory and cardinality arguments. We will use these results in Sec. V to re-write UMMs, quantum computers and liquid-state machines in this set-theory language. This allows us to show explicitly in Sec. VI that UMMs are not only Turing-complete, but also quantum-complete and liquid-state complete. In Sec. VII we offer our conclusions and thoughts for future work.

II. Review of Machine Definitions

We first provide a brief review of the definitions of the three machines we will be discussing in this paper, so the reader will have a basis of reference.

A. Universal Memcomputing Machines

The UMM is an ideal machine formed by interconnected memory cells ("memcells" for short or memprocessors). It is set up in such a way that it has the properties we have anticipated of intrinsic parallelism, functional polymorphism, and information overhead [4].

We can define a UMM as the eight-tuple [4]:

$$(M, \Delta, P, S, \Sigma, p_0, s_0, F) \tag{1}$$

where M is the set of possible states of a single memory cell, and Δ is the set of all transition functions:

$$\delta_a : M^{m_a} \setminus F \times P \to M^{m'_a} \times P^2 \times A \tag{2}$$

where m_a is the number of cells used as input (being read), F is the final state, P is the set of input pointers, m'_a is the number of cells used as output (being written), P^2 is the Cartesian product of the set of output pointers and the set of input pointers for the next step, and A is the set of indices a.

Informally, the machine does the following: every transition function has some label a, and the chosen transition function directs the machine to what cells to read as inputs. Depending on what is being read, the transition function then writes the output on a new set of cells (not necessarily distinct from the original ones). The transition function also contains information on what cells to read next, and what transition function(s) to use for the next step.

Using this definition, we can better explain what the two properties of intrinsic parallelism and functional polymorphism mean. Unlike a Turing machine, the UMM does not operate on each input cell individually. Instead, it reads the input cells as a whole, and writes to the output cells as a whole. Formally, what this means is that the transition function cannot be written as a Cartesian product of transition functions acting on individual cells, namely $\delta(\prod_i M_i) \neq \prod_i(\delta_i(M_i))$. This is the property of intrinsic parallelism. We will later show that the set of transition functions without intrinsic parallelism is, in fact, a small subset of the set of transition functions with intrinsic parallelism.

Furthermore, the transition function of the UMM is dynamic, meaning that it is possible for the transition function to change after each time step. This is shown as the set A at the output of the transition function, whose elements indicate what transition functions to use for the next time step. This is the property of functional polymorphism, meaning that the machine can admit multiple transition functions. Finally, the topology of the network is encoded in the definition of the transition functions which map a given state of the machine into another. A more in depth discussion of these properties can be found in the original paper [4].

B. Liquid-State Machines

Informally, we can think of the LSM as a reservoir of water [16]. The process of sending the input signals into the machine is analogous to us dropping stones into the water, and the evolution of the internal states of the machine is the propagation of the water waves. Different waveforms will be excited depending on what stones were being dropped, how and when they were dropped, and the properties of the waveforms will encompass the information of the stones being dropped. Therefore, we can train a function that maps the waveforms to the corresponding output states that we desire.

Formally, we can define the machine using a set of filters and a trained function [13]. A series of filters defines the evolution of the internal states, and the trained function maps the internal states to some output.

The set of filters must satisfy the point-wise separation property. This is defined as follows:

Definition 1. *A class B of basis filters has the* pointwise separation property *if for any two input functions u and v, with $u(s) \neq v(s)$ for some $s \leq t$, there is a basis filter $b \in B$ such that $(b \circ u)(t) \neq (b \circ v)(t)$.*

This means that we can choose a series of filters such that the evolution of the internal states will be unique to any given signal, at any given time. In other words, this property ensures that different "stones" excite different "waveforms".

The trained output function must satisfy the "fading memory" property. This is defined as follows:

Definition 2. *$F : U \to \mathbb{R}^n$ has* fading memory *if for every internal state $u \in U$ and every $\epsilon > 0$, there exist $\delta > 0$ and $T > 0$ so that $|(Fv)(0) - (Fu)(0)| < \epsilon$ for all $v \in U$ with $|u(t) - v(t)| < \delta$ for all $t \in [-T, 0]$.*

Intuitively, this means that we do not need to know the evolution of the internal states from the infinite past to determine a unique output. Instead, we only need to know the evolution starting from a finite time $-T$.

C. Quantum Computers

There are many ways in which one can define a quantum computer. For simplicity sake, we consider the most general model of quantum computing and ignore its specific operations.

Consider a quantum computer with n identical qubits, and each qubit can be expressed as a linear combination of m basis states. The choice of the basis states can be arbitrary. However, they have to span the entire Hilbert space of the system. If we look at one single qubit, then every state that it admits can be expressed in terms of a linear combination of the m basis states. (Typically m is chosen equal to 2, but we do not restrict the quantum machine to this basis number here.) In other words, $|\psi\rangle = \sum_{i=0}^{m-1} c_i |i\rangle$, where i simply labels the basis state, and c_i is some complex number. Note that we have to impose the normalization condition $\sum_{i=0}^{m-1} |c_i|^2 = 1$.

Now, let us consider the whole system. In general, the total state can be expressed as $|\psi\rangle = \sum_{i_1=0}^{m-1} \sum_{i_2=0}^{m-1} \cdots \sum_{i_n=0}^{m-1} c_{i_1 i_2 \ldots i_n} |i_1 i_2 \ldots i_n\rangle$. Here, i_j denotes that the j-th qubit is in the i-th basis state. A basis state of the total wavefunction can be expressed as the tensor product of the basis states of the individual qubits. Then it is not hard to see that the total state will have a total number of m^n basis states. Each basis state is associated with some complex factor $c_{i_1 i_2 \ldots i_n}$ where, again, the normalization condition $\sum_{i_1=0}^{m-1} \sum_{i_2=0}^{m-1} \cdots \sum_{i_n=0}^{m-1} |c_{i_1 i_2 \ldots i_n}|^2 = 1$ has to be imposed.

Any quantum algorithm can be expressed as a series of unitary operations [17]. Since the product of multiple unitary operations is still a unitary operation, we can then express the total operation after time t with a single operator $\hat{U}(t)$, so that $|\psi(t)\rangle = \hat{U}(t) |\psi(0)\rangle$. In other words, the state of the quantum system at any point in time can be expressed as some unitary operation on the initial state. Note that $\hat{U}(t)$ can be either continuous or discrete in time. Either way, $\hat{U}(t)$ can be considered as the "transition function" of the quantum computer.

Finally, we have to make measurements on the system in order to convert the quantum state into some meaningful *output* for the observable we are interested in. The process of measurements can be considered as finding the expectation value of some observable $\hat{O}$, so the *output function* of a quantum computer can be written as $\langle\psi(t)|\hat{O}|\psi(t)\rangle = \langle\psi(0)|\hat{U}(t)^\dagger \hat{O} \hat{U}(t)|\psi(0)\rangle$. Of course, to obtain an accurate result of the expectation value, many measurements have to be made. In fact, the initial state has to be prepared multiple times, evolved multiple times, and the corresponding expectation value at a given time, must be measured multiple times. A quantum computer is thus an intrinsically probabilistic-type of machine.

III. Mathematical Tools

After the description of the three machines we consider in this work, we now introduce the necessary mathematical tools that will allow us to construct a general model of a computing machine using set theory and cardinality arguments. Most of the definitions and theorems in this section, with their detailed proofs can be found in the literature on the subject (see, e.g., the textbook [18]).

We denote the cardinality of the set of all natural numbers as $\aleph_0$ ($|\mathbb{N}| = \aleph_0$), and the cardinality of the set of all real numbers as $\mathfrak{c}$ ($|\mathbb{R}| = \mathfrak{c}$). Then, one can prove the following theorems [19]:

Theorem III.1. *The power set of natural numbers has cardinality $\mathfrak{c}$, $|2^{\aleph_0}| = 2^{|\aleph_0|} = \mathfrak{c}$.*

Theorem III.2. *Any open interval of real numbers has cardinality $\mathfrak{c}$.*

We can generalize the concept of infinity by introducing Beth numbers [20], defined as follows:

Definition 3. *Let $\beth_0 = \aleph_0$, and $\beth_{\alpha+1} = 2^{\beth_\alpha}$ for all $\alpha \in \mathbb{N}$.*

By this definition, we see that $\mathfrak{c} = \beth_1$ and $2^{\mathfrak{c}} = \beth_2$. Each Beth number is strictly greater than the one preceding it. The following theorem [18] allows us to perform arithmetics on infinite cardinal numbers, and derive relationships between Beth numbers.

Theorem III.3. *Given any two numbers, μ and κ, if at least one of them is infinite, then $\mu + \kappa = \mu\kappa = \max\{\mu, \kappa\}$.*

Using this theorem, we can prove the following properties of Beth numbers:

Corollary III.3.1. *$\beth_\beta \beth_\alpha = \beth_\beta + \beth_\alpha = \beth_\beta$ for all $\alpha, \beta \in \mathbb{N}$, where $\alpha \leq \beta$.*

Corollary III.3.2. *$\mu^{\beth_\alpha} = \beth_{\alpha+1}$ if $2 \leq \mu \leq \beth_{\alpha+1}$. $(\beth_\alpha)^\kappa = \beth_\alpha$ if $1 \leq \kappa \leq \beth_{\alpha-1}$*

Proof. Corollary III.3.1 can be proven trivially if we note that each Beth number is greater than its predecessor. □

Proof. Corollary III.3.2 can be proven as follows. By definition $2^{\beth_\alpha} = \beth_{\alpha+1}$. Furthermore, $(\beth_{\alpha+1})^{\beth_\alpha} = (2^{\beth_\alpha})^{\beth_\alpha} = 2^{(\beth_\alpha \beth_\alpha)} = 2^{\beth_\alpha} = \beth_{\alpha+1}$, where $\beth_\alpha \beth_\alpha = \beth_\alpha$ from Corollary III.3.1. Since $2 \leq \mu \leq \beth_{\alpha+1}$, then $\beth_{\alpha+1} = 2^{\beth_\alpha} \leq \mu^{\beth_\alpha} \leq (\beth_{\alpha+1})^{\beth_\alpha} = \beth_{\alpha+1}$. This implies $\mu^{\beth_\alpha} = \beth_{\alpha+1}$. The second equality can be proven in the same vein. □

In the following section, we will define computing models using Cartesian products and mapping functions. The following two theorems will be helpful [18]:

Theorem III.4. *Let the set S be the Cartesian product of the sets S_1 and S_2, or $S = S_1 \times S_2$. Then $|S| = |S_1||S_2|$.*

Theorem III.5. *Let $f : S \to T$ be a function that maps set S to set T, and let F be the set of all possible functions f. Then $|F| = |T|^{|S|}$.*

Finally, we introduce the following theorem, which can be derived directly from the definition of cardinality [18]:

Theorem III.6. *Two sets have the same cardinality if and only if there is a bijection between them.*

This theorem implies that there is a bijection between any two real coordinate spaces regardless of their dimensions:

Corollary III.6.1. *There is a bijection between $\mathbb{R}^n$ and $\mathbb{R}^m$, for any $n, m \in \mathbb{N}$.*

Proof. From Corollary III.3.2, we see that $|\mathbb{R}^n| = |\mathbb{R}|^n = (\beth_1)^n = \beth_1$. Similarly, $|\mathbb{R}^m| = \beth_1$. Therefore, the two sets have the same cardinality, so there is a bijection between the two. □

Note that for complex coordinate spaces, $|\mathbb{C}^n| = |\mathbb{C}|^n = |\mathbb{R}^2|^n = |\mathbb{R}|^{2n} = (\beth_1)^{2n} = \beth_1$. Therefore, there is a bijection between any two complex coordinate spaces as well. Furthermore, there is a bijection between any complex coordinate space and any real coordinate space.

IV. General Computing Model

We have now introduced all the mathematical tools necessary to define a computing machine using set theory. We begin by describing the Cartesian product of two sets - the set of all internal states and the set of all transition functions.

A. Internal States

Consider a general computing machine. We let the state variable s describe the full internal state of the machine, which belongs to a set S of states. The internal state should encompass all the necessary information, such that given this state variable s and any transition function δ (which will be defined shortly), the machine can fully determine the next internal state. The important thing to note is that not all the elements of S are necessarily possible internal states of the machine. In other words, the set of all possible internal states is only a subset of S. The set S and this subset are however not necessarily equivalent. The reason for this will be explained in greater detail later in Sec. IV-C.

As an example, consider the full internal state of the Turing machine. The internal state should include the Cartesian product of three states - the register state of the control ($s^{(1)}$), the tape symbols written on the cells ($s^{(2)}$), and the current address of the head ($s^{(3)}$) [21]. Given these three states and some transition function (which depends on how the machine is coded), then the processor will know what to write (thereby changing $s^{(2)}$), in what direction to move (thereby changing $s^{(3)}$), and what new state to be in (thus changing $s^{(1)}$).

We see that the set S is expressible as a Cartesian product of three sets, $S = S^{(1)} \times S^{(2)} \times S^{(3)}$, then $|S| = |S^{(1)}||S^{(2)}||S^{(3)}|$. Consider a Turing machine with m tape symbols, n tape cells, and k register states. It is easy to see that $|S^{(1)}| = k$, $|S^{(2)}| = m^n$, and $|S^{(3)}| = n$. Therefore, we can calculate $|S| = |S^{(1)}||S^{(2)}||S^{(3)}| = km^n n < |\mathbb{N}| = \beth_0$. The strict inequality comes from the fact that m, n, and k are all finite numbers. In other words, this is a finite digital/discrete machine, and in general, the $|S| < \beth_0$ is true for a finite digital machine.

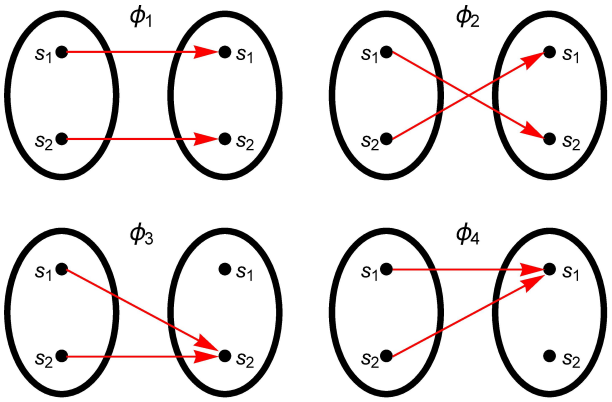

Fig. 1. The set of all transition functions for $|S| = 2$. There are $2^2 = 4$ transition functions in total. A transition function on this set is described by two red arrows.

On the other hand, if we consider the theoretical model of a Turing machine with infinite tape cells μ, then the calculation changes significantly. First, note that $\mu = |\mathbb{N}| = \beth_0$ because we are working with infinite discrete cells and we can map each cell to an integer in $\mathbb{N}$. Then, we have $|S^{(1)}| = k$, $|S^{(2)}| = m^{\beth_0} = \beth_1$ (from Corollary III.3.2), and $|S^{(3)}| = \beth_0$. Therefore, we see that $|S| = |S^{(1)}|S^{(2)}||S^{(3)}| = k\beth_1\beth_0 = \beth_1$ (from Corollary III.3.1). For the purpose of proving Turing-completeness, we use this model of infinite tape.

B. Transition Function

1) General Definition of Transition Function: The operation of any computing machine is defined using a transition function [21] (or a set of transition functions such as in a general UMM [4]). However the transition functions we consider are defined as "deterministic" in contrast to the "non-deterministic" transition functions used for example to define the non-deterministic Turing Machine [3]. Here, we give a much more general definition of the (deterministic) transition function. Essentially, we are throwing away constraints such as initial states, accepting states, tape symbols, and so forth, and simply define the transition function as a mapping from some state to some other (or the same) state. We can then formally define the transition function as follows:

Definition 4. *Let S be any set, then φ_S is said to be a* transition function *on S if, for every $s \in S$, we have a unique $\varphi_S(s) \in S$. In other words, φ_S is a function that maps S to itself, or $\varphi_S : S \mapsto S$. We denote the set of all transition functions on S as Φ_S.*

From Theorem III.5, it is easy to see that the cardinality of Φ_S is simply $|\Phi_S| = |S|^{|S|}$. Again, it is important to note that the set of the *actual* transition functions of a given machine is a subset of all *possible* transition functions Φ_S. The two are generally not equivalent. This is because we are not defining

the transition function based on *the operation of the machine*. Instead, we are defining the transition function as the *mapping of a set to itself* (see Fig. 1 for a schematic representation), so there are transitions that are impossible for the machine to support. The difference between the "possible" and "actual" transition functions will be formalized in section IV-C.

2) Turing Machine as Example: For a Turing machine, after the machine is coded, the transition function remains stationary and cannot be changed during the execution of an algorithm (unlike a UMM where the transition function is dynamic). In other words, the machine will take some initial internal state s_i and apply some transition function φ_S to it recursively until the final state s_f is reached and the machine halts. We can express this as $s_f = \underbrace{\varphi_S(...\varphi_S(\varphi_S}_{\text{n iterations}}(s_i)))$, where n is some integer. Furthermore, when the final state is reached, we should have $s_f = \varphi_S(s_f)$, meaning that the transition function should not alter the final state, and this represents the termination of the algorithm.

The entire machine process can be fully described given some initial state s_i and some transition function φ_S. To show this, we only have to consider a single step process, or show that there is an appropriate choice of φ_S such that given some state s, $\varphi_S(s)$ will always give us the expected next state for any algorithm. The following argument will demonstrate why this is true.

From section IV-A, we know that given a state s_i, we can divide the state into three components, $s_i = s_i^{(1)} \times s_i^{(2)} \times s_i^{(3)}$. Recall that the three components give the register state, the tape symbols on the tape cells, and the address of the head respectively. First, the machine reads the tape symbol under the head. This is equivalent to the transition function taking the input $s_i^{(3)}$ (locating the head) and $s_i^{(2)}$ (reading the tape symbol). Then, according to what is being read and the register state of the control, the head writes a new symbol on the tape cell. This corresponds to the transition function taking the input $s_i^{(1)}$ (reading the register state) and outputting $s_{i'}^{(2)}$ (updating the tape symbol). Finally, the machine moves the head and changes the register state. This is equivalent to the transition function outputting $s_{i'}^{(1)}$ (updating the register state) and $s_{i'}^{(3)}$ (updating the address). Therefore, we see that a single state machine process is fully encompassed in the transition $s_i \to s_{i'}$. For every $s_i \in S$, there exists a unique $s_{i'} \in S$ associated with it. Then we can choose $\varphi_S \in \Phi_S$ such that $\varphi_S(s_i) = s_{i'}$ is satisfied for every $s_i \in S$. Therefore, we see that φ_S fully describes the single step machine process, and this implies that recursions of φ_S can describe any Turing machine algorithm.

Even though one can find a $\varphi_S \in \Phi_S$ for every Turing machine algorithm, the reverse is not true. In fact, there are transition functions in Φ_S that the Turing machine cannot support. For example, consider the operation of simultaneously writing two tape symbols on two cells. This cannot be done by the Turing machine since by definition, a Turing machine can only write one tape symbol on one tape cell at a time. However, this transition function is not excluded *a priori* from Φ_S, because we can always find a $\varphi_S \in \Phi_S$, such that $s_i^{(2)}$ and $s_{i'}^{(2)} = \varphi_S(s_i^{(2)})$ differ by two tape symbols for some $s_i \in S$.

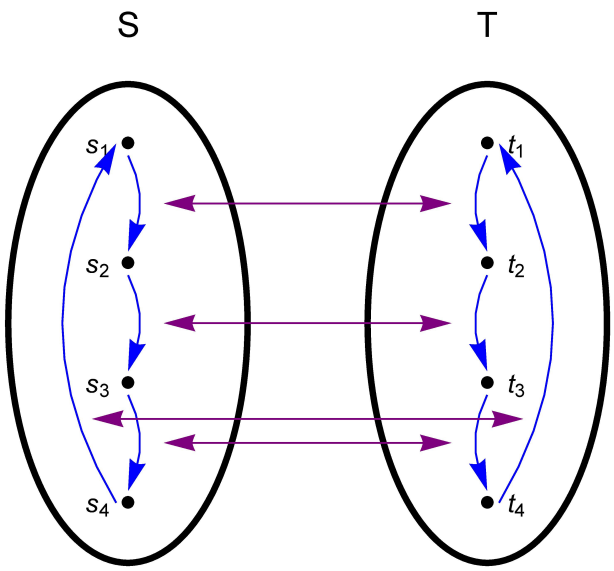

Fig. 2. Let S and T be two sets of the same cardinality, and label the set elements such that s_i pairs with t_i. The blue arrows represent a single transition function for each set. Informally, the pairing of the two transition functions under the "proper bijection" is represented by the purple arrows that pair the arrows in set S with their respective counterparts in set T.

3) Proper Bijection: Let us first consider the following important theorem:

Theorem IV.1. *If* $|S_1| = |S_2|$, *then* $|\Phi_{S_1}| = |\Phi_{S_2}|$.

The theorem can be easily proven using cardinality arguments (from Theorem III.6). In other words, if there exists a bijection between two sets, then there must also be a bijection between the sets of their respective transition functions.

Note that at this point, we have only proved that there is a bijection between the two transition function sets. However, in our case, we are really only interested in one particular bijection. For simplicity, let us define this bijection for the $|S| = \beth_1$ case. This definition will easily generalize to cases where $|S| > \beth_1$.

If $|S| = \beth_1$, then there is a bijection from S to $\mathbb{R}$. Denote this bijective function as $\mathbb{R} = f(S)$. Let $\varphi_S \in \Phi_S$ be some transition function on S, and for $\forall s_i \in S$, let $s_j = \varphi_S(s_i)$, where $i, j \in \mathbb{R}$ are just arbitrary real number labels. Then we can define a bijection from S to $\mathbb{R}$ such that $f(s_i) = i$ and $f(s_j) = j$. Note that if we let $\varphi_{\mathbb{R}} = f \circ \varphi_S \circ f^{-1}$, then $j = \varphi_{\mathbb{R}}(i)$. Now let T be another set such that $|T| = |S|$, then similarly, we can let $g(t_i) = i$ and $g(t_j) = j$. Now if we let $\varphi_T = g^{-1} \circ \varphi_{\mathbb{R}} \circ g$, then $t_j = \varphi_T(t_i)$. We can then define φ_T as the *proper bijection* of φ_S, and we do this for every single element of the transition function set.

To put this informally, we first define some bijection from set S to set T, then if s bijects to t, then $\varphi_S(s)$ also bijects to $\varphi_T(t)$. The proper bijection essentially maps all machine operations from one machine to another machine. See Fig. 2 for a schematic representation of a pairing of transition functions under the proper bijection.

The following theorem is a restatement of the above dis-

cussion:

Theorem IV.2. *If $|S| = |T|$, then there is a proper bijection from Φ_S to Φ_T.*

It is easy to see that a proper bijection implies that any algorithm that the machine $S \times \Phi_S$ can run, can also be run by machine $T \times \Phi_T$. This essentially means that S and T are *equivalent* [22]. However, this is under the assumption that there are absolutely no *constraints* whatsoever on the transition functions and/or states of the machines. As we will see shortly, this is generally not the case.

C. Constraints

The above naive argument essentially states that if the internal states of two different machines have the same cardinality, then the two machines are equivalent. This is obviously not the case. Any computational model has always some *constraints*, either on the internal states or the transition functions (or both) of the machine at hand.

In other words, the *actual* set of internal states S' is a subset of S, and the *actual* set of transition functions Φ'_S is a subset of Φ_S. In some sense, the computational structure of a machine is defined by its constraints, so we should carefully discuss what constraints actually are.

1) Constraints on Internal States: We make the following distinction between the *full* set of internal states and the *actual* set of internal states.

Definition 5. *Let S' be the set of all possible internal states that a machine can support. We call this the* actual *set of internal states. Let S be some arbitrary superset of S'. We call S the* full *set of internal states.*

As an example, consider a very simple computer that only consists of two bits, with each bit supporting two states, 0 and 1. However, the two bits are connected in such a way that they must have the same state at a given time. In other words, the actual set of internal states is $S' = \{00, 11\}$, where ij denotes the first bit being in state i and the second bit being in state j.

A natural choice for the full set is instead $S = \{00, 01, 10, 11\}$, namely the set of internal states if the two bits were not connected (constrained). However, we may as well have chosen S to be, for example, $S = \{00, 01, 10\}$ or $S = \{00, 01, 10, 11, -10\}$. (Note that the last element of the second set, -10, represents a state that is not supported by the machine, even if the constraint were to be removed.) However, it is usually convenient for us to choose the full set such that it includes, and only includes, all the possible states when the constraints on the machine are removed.

2) Constraints on transition function: Following a similar reasoning, we distinguish between the *full* set of transition functions and the *actual* set of transition functions. The full set of transition functions on S is already defined in Def. 4, and the definition of the actual set is as follows:

Definition 6. *Let Φ'_S be the set of all transition functions of a machine when the constraints on the machine are included. We call this the* actual *set of transition functions.*

Following the above example, $S = \{00, 01, 10, 11\}$ and $S' = \{00, 11\}$. For convenience, let us express the set elements in decimal representation. Then $S = \{0, 1, 2, 3\}$ and $S' = \{0, 3\}$.

We can then choose Φ'_S so that it only contains 4 transition functions, and we let them to be $\varphi_{S1}(n) = n$, $\varphi_{S2}(n) = (n+1) \pmod 4$, $\varphi_{S3}(n) = (n+2) \pmod 4$, and $\varphi_{S4}(n) = (n+3) \pmod 4$. Note that this is obviously *not* the full set of transition functions because, for example, there is no transition function such that $\varphi_{Si}(0) = 2 \wedge \varphi_{Si}(2) = 3$ (where $\wedge$ is the logical AND) is satisfied. In fact, the full set of transition functions has cardinality of $4^4 = 256$, and we merely considered 4 of them.

In addition, the transition functions Φ'_S and $\Phi_{S'}$ are *not* the same. $\Phi_{S'}$ is the *full* set of transition functions on the *actual* set of internal states S'.

To illustrate this difference, consider the previous example of $S' = \{0, 3\}$. Let us try to find the full set of transition functions on this set. There should be $2^2 = 4$ possible transition functions. They are $\varphi_{S'1}(n) = n$, $\varphi_{S'2}(n) = 3 - n$, $\varphi_{S'3}(n) = 3$, and $\varphi_{S'4}(n) = 0$. Without any formal proof, we can already see that $\Phi_{S'}$ is very different from Φ'_S, and there is no proper bijection between Φ'_S and $\Phi_{S'}$ even though they have the same cardinality of 4. This is because the sets that the transition functions act on are different (they have different cardinality), so it makes no sense to pair them together.

An important aside is that the transition function is defined by its operation on every element of a particular set and not its *analytic* representation. For example, $\varphi_{S'}(n) = 3 - n$ is different from $\varphi_S(n) = 3 - n$ even though they have the same analytic expression. (For example, $\varphi_{S'}(2)$ is not defined while $\varphi_S(2) = 1$.) On the other hand, $\varphi_{S'i}(n) = 3 - n$ and $\varphi_{S'j}(n) = 3 - \frac{n^2}{3}$ are the same on set S' even though their analytic expressions are different. (They are, however, not the same on set S.)

D. Equivalence and Completeness

At this point we can recall the traditional definitions of "equivalence" and "completeness" among machines. The equivalence relationship is defined as follows [22]:

Definition 7. *Two machines, $S \times \Phi'_S$ and $T \times \Phi'_T$, are said to be* equivalent *if $S \times \Phi'_S$ can simulate $T \times \Phi'_T$, and $T \times \Phi'_T$ can simulate $S \times \Phi'_S$.*

Then the following theorem is clearly true:

Theorem IV.3. *Two machines, $S \times \Phi'_S$ and $T \times \Phi'_T$, are equivalent if $|S| = |T|$, and there is a proper bijection between Φ'_S and Φ'_T.*

This theorem is fairly obvious. If there is a proper bijection between two sets of transition functions, then we can map every machine operation from the first machine to the second machine, and vice versa. This implies that the two machines can simulate each other. Note that from Section IV-B3, it is clear that there is always a proper bijection between the two full sets of transition functions Φ_S and Φ_T if $|S| = |T|$, so if two equal-sized machines admit the full sets of transition functions, then the two machines are equivalent.

Now, the natural next step is to define the concept of completeness, but before we do so, let us first introduce the concept of a *reduced machine*.

Definition 8. *Consider a machine $S \times \Phi'_S$. If $S' \subset S$, then we call $S' \times \Phi'_{S'}$ the* reduced machine *of $S \times \Phi'_S$.*

It is easy to obtain mathematically a reduced machine from a full machine. First, we let $S'' = S \backslash S' = \{s | s \in S \wedge s \notin S'\}$, then we can express the full set as $S = S' \cup S''$ (note this is only true if $S' \subseteq S$). It is not hard to show that $\Phi'_{S'} \times \Phi'_{S''} \subseteq \Phi'_S$, and from this subset we can simply "ignore" the $\Phi'_{S''}$ factor. Then it is clear that $S \times \Phi'_S$ simulates $S' \times \Phi'_{S'}$, but not the other way around. Informally, we are using a machine with greater "resources" to simulate another machine with lesser "resources".

Let us then recall the conventional definition of completeness [22]:

Definition 9. *Machine, $S \times \Phi'_S$, is $T \times \Phi'_T$-*complete *if the former can simulate the latter.*

From this definition, it is clear that, for the example above, $S \times \Phi'_S$ is $S' \times \Phi'_{S'}$-complete. Note also that equivalence implies completeness, but the reverse is not true.

It is also obvious that $S \times \Phi_S$ can simulate $S \times \Phi'_S$, since Φ'_S is just a subset of the full set of machine processes (transition functions). Then we can also say that $S \times \Phi_S$ is $S \times \Phi'_S$-complete. Informally, we are using an unconstrained machine to simulate a constrained machine.

Let us then recall a few obvious lemmas:

Lemma IV.4. *If machine A is equivalent to machine B, and machine B is equivalent to machine C, then machine A is equivalent to machine C.*

Lemma IV.5. *If machine A is B-complete, and machine B is C-complete, then machine A is C-complete.*

With these lemmas and preliminaries, we can then prove this important theorem:

Theorem IV.6. *Machine $S \times \Phi'_S$ is $T \times \Phi'_T$-complete if $|T| \leq |S|$, and if we can find a reduced machine $S' \times \Phi'_{S'}$ such that $|S'| = |T|$ and there is a proper bijection between Φ'_T and some subset of $\Phi'_{S'}$.*

Proof. The proof of this is quite simple. We know that $S \times \Phi'_S$ is $S' \times \Phi'_{S'}$-complete, since the latter is just the reduced machine of the former. Furthermore, we also know that $S' \times \Phi'_{S'}$ is $T \times \Phi'_T$-complete. This is because Φ'_T is just a subset of $\Phi'_{S'}$, or there is a "proper injection" from Φ'_T to $\Phi'_{S'}$, implying that any machine process of $T \times \Phi'_T$ can be simulated by $S' \times \Phi'_{S'}$. Therefore, from Lemma IV.5, we see that $S \times \Phi'_S$ is $T \times \Phi'_T$ complete. □

From this theorem, we can immediately derive a corollary that will be useful for showing the universality of memcomputing machines in section VI:

Corollary IV.6.1. *Machine $S \times \Phi_S$ is $T \times \Phi'_T$-complete if $|T| \leq |S|$, where Φ_S is the full set of transition functions and Φ'_T can either be the actual set or the full set of transition functions.*

Proof. First, we find a reduced machine $S' \times \Phi_{S'}$ of $S \times \Phi_S$ such that $|S'| = |T|$. Note that if a machine admits the full set of transition functions, then its reduced machine also admits the full set. Since $|S'| = |T|$, then we can find a proper bijection from $\Phi_{S'}$ to Φ_T (since both sets of transition functions are full). Under the same proper bijective mapping, we can map Φ'_T (a subset of Φ_T) to $\Phi'_{S'}$ (a subset of $\Phi_{S'}$). In other words, there is a proper bijection from Φ'_T to some subset of $\Phi_{S'}$, and from Theorem IV.6, machine $S \times \Phi_S$ is $T \times \Phi'_T$-complete. □

V. Revised Machine Definitions within Set Theory

Up to this point, we have avoided the discussion of the concept of "output states" of a machine. However, under this new mathematical framework, this concept is easily expressible.

In general, the internal state of the machine must be "decoded" into an output state to be read by the user. We can denote the set of all possible output states as F. Then it is obvious that $|F| \leq |S'|$, otherwise we would not be able to find a function that maps S' to F. In other words, every internal state must correspond to a unique output state, and it is easy to show that the output function is expressible as a transition function.

To show this, we first choose a subset $S'_F \subseteq S'$ such that $|S'_F| = |F|$, then there is a bijection between F and S'_F. Therefore, we can simply describe the output mapping function as a transition function that maps S' to S'_F. Then it is clear that the set of all output mapping functions is a subset of $\Phi_{S'}$, so there is no need to redefine a new set to include the output functions.

The mathematical framework has now been fully established, and we are ready to redefine the three machines we are considering in this work within this framework.

A. Universal Memcomputing Machines

In the original definition of the UMM, there is the complication of input and ouput pointers (see Eq. (1)). Within the new set-theory framework, we can avoid the concept of pointers, since the transition function always reads the full internal state (all the cells) and writes the full internal state. In other words, we consider the combination of all cell states as a whole, and make no effort in describing which cell admits which state. In this case, intrinsic parallelism is obviously implied.

Let us then discuss the cardinality of the set of internal states. For a UMM, there is a finite number of cells, $n < \beth_0$, and each cell may admit a continuous state (with cardinality $\beth_1$). In this case, it is easy to show that $|S| = (\beth_1)^n = \beth_1$.

Furthermore, in the original definition of the UMM [Eq. (1)], there are no constraints on the transition function δ. This means that we can use the full set of transition functions, Φ_S, to describe the machine. In this case, functional polymorphism is obviously implied. Therefore, we can define the UMM machine as $S \times \Phi_S$, with $|S| = \beth_1$.

B. Liquid-State Machine

It is not hard to see that the internal state structures for the LSM and the UMM are similar. Instead of memory cells, the

LSM has neurons. But if we make the conservative assumption that there are no constraints on the internal states of the LSM, then the cardinality of the set of internal states for this machine is the same as that of a UMM, $|S| = \beth_1$. However, what real distinguishes the LSM from the UMM is that the set of transition functions for the LSM is not full.

Recall that the LSM consists of a series of filters and an output function (see Sec. II). The set of filters satisfies the point-wise separation property, and the function satisfies the fading-memory property. There is no need to express the two properties in the language of our new framework. Instead, it is enough to note that the point-wise property is a property of a set, while the fading-memory property is a property of an element of the set.

Therefore, we can find a subset $(\Phi_S)_1 \subseteq \Phi_S$ such that its elements represent the filters, with the subset itself satisfying the point-wise separation property. As discussed earlier, we can express the output function as a transition function, so we can find a subset $(\Phi_S)_2 \subseteq \Phi_S$ such that its elements represent the output functions, and they satisfy the fading-memory property. Then, we can take the union of the two subsets $\Phi'_S = (\Phi_S)_1 \cup (\Phi_S)_2$ to get the actual set of transition functions on S.

Therefore, we can describe the LSM as $S \times \Phi'_S$, with $|S| = \beth_1$. The specific structure of the machine can be defined by expressing the two properties as constraints on Φ'_S. This is a slightly tedious process, so we will not be presenting it here, since it is irrelevant for our conclusions.

C. Quantum Computers

Again, consider a quantum computer with n identical qubits, each having m basis states. In subsection II-C, we have shown that the total number of basis states for the entire system is $m^n < |\mathbb{N}|$. Each basis state is associated with some complex factor $c_{i_1 i_2 \ldots i_n} \in \mathbb{C}$. And these factors are constrained by the normalization condition $\sum_{i_1=0}^{m-1} \sum_{i_2=0}^{m-1} \cdots \sum_{i_n=0}^{m-1} |c_{i_1 i_2 \ldots i_n}|^2 = 1$. Furthermore, in practice we usually ignore an overall phase factor since it does not affect the expectation value of an observable.

At this point, it is clear that we can fully describe a quantum state as the Cartesian product of the complex factors for all basis states. In other words, we can represent an internal state as $c_{0_1 0_2 \ldots 0_n} \times c_{1_1 0_1 \ldots 0_n} \cdots \times c_{(m-1)_1 (m-1)_2 \ldots (m-1)_n}$, where there are m^n factors.

Given this information, we can calculate the cardinality of the full set of internal states to be $|S| = |\mathbb{C}^{m^n}| - |\mathbb{R}^2| = |\mathbb{C}^{m^n}| \leq |\mathbb{C}^{\mathbb{N}}| = |\mathbb{C}|^{|\mathbb{N}|} = (|\mathbb{R}|^{|\mathbb{N}|})^2 = (\beth_1^{\beth_0})^2 = (\beth_1)^2 = \beth_1$. (The unimportant $|\mathbb{R}^2|$ is from the normalization condition and factoring out the overall phase factor.) In addition, $|S| = |\mathbb{C}^{m^n}| \geq |\mathbb{R}| = \beth_1$. Therefore, we have $|S| = \beth_1$.

The full set of internal states contains quantum states with varying degrees of entanglement. It is worth stressing though that, in practice, it is extremely hard to construct a quantum computer that can support the full set of quantum states. For example, it is very challenging to prepare 100 qubits that are fully entangled. (The current record is on the order of tens of fully-entangled qubits [23][24].) Therefore, the actual set S' is a small subset of S, or $S' \subseteq S$, unless m and n are both very small. However, since we are making here only theoretical arguments, let us just assume that the full set S of all possible entangled states can be supported.

As discussed previously, the transition functions of a quantum computer can be expressed as unitary operations on some initial state. The set of all unitary operations is obviously a strict subset of the full set of transition functions. For example, you cannot find an unitary operation that collapses every single state to $|00...0\rangle$ (setting $c_{0_1 0_2 \ldots 0_n} = 1$, and setting all the other factors to 0), though this is included in Φ_S. Therefore, we can describe the quantum computer as $S \times \Phi'_S$.

A few words on the "output function" of a quantum computer are also in order. The output function essentially represents the operation of taking the expectation value of some observable on the internal state, or $\langle \psi | \hat{O} | \psi \rangle$. This maps the set of internal states to the set of output states $F \subseteq \mathbb{R}$ (expectation values have to be real), so we obviously have $|F| \leq |S|$.

VI. Universality of Memcomputing Machines

From the above discussions, we can summarize all the results we have obtained so far, and express all the machines we considered here in their most general form:

- Turing Machine: $T' \times \Phi'_{T'}$, $|T'| \leq \beth_1$,
- Liquid-State Machine: $L \times \Phi'_L$, $|L| = \beth_1$,
- Quantum Computer: $Q \times \Phi'_Q$, $|Q| = \beth_1$,
- Universal Memcomputing Machine: $M \times \Phi_M$, $|M| = \beth_1$.

Therefore, by applying Corollary IV.6.1, we see that a UMM can simulate any Turing machine, any liquid-state machine, and any quantum computer.

Let us expand on this for each pair of machines separately. In particular, let us briefly discuss how a mapping between a UMM and the three other machines can be realized in theory. Of course, this mapping does not tell us anything about the resources required for a UMM to simulate the other machines. Hence, this is by no means a discussion on how to realize an efficient or practical mapping.

1) UMM vs. Turing Machines: Let us look at the mapping between a Turing machine and a UMM. First, we map each tape cell to a memory cell (memcell). We can denote these memcells collectively as a "memtape". The tape symbols can be mapped to the internal states of each memcell of the memtape.

Then, we can map the state register to another memcell which we will denote as "memregister". The state of the Turing machine is then stored as the internal state of the memregister. Finally, we can store the current address of the head as an internal state of yet another memcell which we denote as "memaddress".

We can then wire the memcells together into a circuit such that it simulates the operation of the Turing machine. Note that as a result of functional polymorphism, we do not have to re-wire the circuit each time we choose to run a different algorithm. The circuit first reads the memregister and the memaddress, so that it knows which memcell of the memtape to modify and how to modify it. After that memcell

is modified, the memregister and memaddress then update themselves to prepare for the next cycle. In short, we are replacing the tape, head, and control with memprocessors.

2) UMM vs. LSM: The mapping between a LSM and a UMM is fairly obvious. We simply have to map each "reservoir cell" to a memcell, and wire the circuit such that the point-wise separation and fading-memory properties are satisfied. The explicit construction of the circuit to realize such properties will not be explored here.

Although, in theory, it is possible to simulate an LSM with a UMM, it is not always efficient or necessary to do so in practice. The circuit topologies of the two machines are very different, and they are designed to perform different tasks.

For the LSM model, the connections between the reservoir cells are typically random, and the reservoir as a whole is not trained. The expectation of getting the correct output relies entirely on training the output function correctly. In the end, the operation of the machine relies on statistical methods, and is inevitably prone to making errors. In some sense, the machine as a whole is analogous to a "learning algorithm" [25].

On the other hand, for the UMM, we can connect the memcells into a circuit specific to the tasks at hand. One realization of this connection employs self-organizing logic gates [5] to control the evolution of the machine such that it will always evolve towards an equilibrium state representing the solution to a particular problem (the machine is *deterministic*).

In the general case, the UMM is an entirely different computing paradigm than the LSM, proven already to provide exponential speed-up for certain hard problems [5], [7]. In other words, while possible, utilizing a UMM to simulate an LSM will not be exploiting all the properties of the UMM to its full use. In practical applications, it would then be more advantageous to use a UMM (and its digital version, a DMM) to tackle directly the same problems investigated by LSMs.

3) UMM vs. Quantum Computers: Simulating a quantum computer with a UMM requires "compressing" the internal state of a quantum computer. Recall that a quantum computer has m^n basis states, and each basis state is associated with some complex factor. All these complex factors would then need to be represented with only $k = O(n)$ *interacting* memcells. In this work we have shown that this is in fact doable on a theoretical level.

However, as already mentioned, this result does not provide any information on the resources required for a UMM to simulate a quantum computer. Nevertheless, one of the features that makes UMMs a practical and powerful model of computation is precisely its "information overhead".

Information overhead and quantum entanglement share some similarities: in some sense, both of them allow the machine to access a set of results of mathematical operations (without actually *storing* them) that is larger than that provided by simply the union of *non-interacting* processing units (cf. Refs. [4] and [8]). We could then argue that we may exploit the information overhead property of a UMM to precisely represent efficiently the entanglement of a quantum system. At this point, however, this question is still open.

VII. Conclusions

In conclusion, we have employed set theory and cardinality arguments to describe the relation between universal memcomputing machines and other types of computing models, in particular liquid-state machines and quantum computers. Using this mathematical framework we have confirmed that UMMs are Turing-complete, a result already obtained in Ref. [4] using a different approach.

In addition, we have also shown that UMMs are liquid-complete (or reservoir-complete), and quantum-complete, namely they can simulate *any* liquid-state (or reservoir-computing) machine and *any* quantum computer. Of course, the results discussed here do not provide an answer to the question of what resources would be needed for a UMM to *efficiently* simulate such machines, only that such a mapping exists. Along these lines, it would be interesting to study the relation between information overhead and quantum entanglement. If such a relation exists and can be exploited at a practical level, it may suggest how to utilize UMMs to efficiently simulate quantum problems that are currently believed to be only within reach of quantum computers (such as the efficient simulation of quantum Hamiltonians). Further work is however needed to address this practical question.

Acknowledgments – MD acknowledges partial support from the Center for Memory and Recording Research at UCSD.

References

[1] M. Di Ventra and Y. V. Pershin, "The parallel approach," *Nature Physics*, vol. 9, pp. 200–202, 2013.

[2] J. v. Neumann, "First draft of a report on the edvac," tech. rep., 1945.

[3] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. New York, NY, USA: Cambridge University Press, 1st ed., 2009.

[4] F. L. Traversa and M. Di Ventra, "Universal memcomputing machines," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 11, p. 2702, 2015.

[5] F. L. Traversa and M. Di Ventra, "Polynomial-time solution of prime factorization and np-complete problems with digital memcomputing machines," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 27, p. 023107, 2017.

[6] F. L. Traversa, C. Ramella, F. Bonani, and M. Di Ventra, "Memcomputing NP-complete problems in polynomial time using polynomial resources and collective states," *Science Advances*, vol. 1, no. 6, p. e1500031, 2015.

[7] F. Traversa, P. Cicotti, F. Sheldon, and M. Di Ventra, "Evidence of an exponential speed-up in the solution of hard optimization problems," *arXiv:1710.09278*, 2017.

[8] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, pp. 1484–1509, Oct. 1997.

[9] A. Finnila, M. Gomez, C. Sebenik, C. Stenson, and J. Doll, "Quantum annealing: A new method for minimizing multidimensional functions," *Chemical Physics Letters*, vol. 219, pp. 343–348, Mar. 1994.

[10] G. H. Low and I. L. Chuang, "Optimal hamiltonian simulation by quantum signal processing," *Physical Review Letters*, vol. 118, Jan. 2017.

[11] P. Benioff, "The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines," *Journal of Statistical Physics*, vol. 22, pp. 563–591, May 1980.

[12] D. Korenkevych, Y. Xue, Z. Bian, F. Chudak, W. G. Macready, J. Rolfe, and E. Andriyash, "Benchmarking quantum hardware for training of fully visible boltzmann machines," 2016.

[13] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[14] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, 2009.

[15] D. Verstraeten, B. Schrauwen, D. Stroobandt, and J. Van Campenhout, "Isolated word recognition with the liquid state machine: a case study," *Information Processing Letters*, vol. 95, no. 6, pp. 521–528, 2005.

[16] H. Jaeger, "The echo state approach to analysing and training recurrent neural networks-with an erratum note," *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, vol. 148, no. 34, p. 13, 2001.

[17] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge Series on information and the Natural Sciences, Cambridge University Press, 10th Aniversary ed., 2010.

[18] H. B. Enderton, *Elements of set theory*. Academic Press, 1977.

[19] G. Cantor, "Ueber eine eigenschaft des inbegriffs aller reellen algebraischen zahlen.," *Journal für die reine und angewandte Mathematik*, vol. 77, pp. 258–262, 1874.

[20] T. E. Forster, "Set theory with a universal set. exploring an untyped universe," 1994.

[21] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *ACM SIGACT News*, vol. 32, no. 1, pp. 60–65, 2001.

[22] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[23] T. Monz, P. Schindler, J. T. Barreiro, M. Chwalla, D. Nigg, W. A. Coish, M. Harlander, W. Hänsel, M. Hennrich, and R. Blatt, "14-qubit entanglement: Creation and coherence," *Physical Review Letters*, vol. 106, no. 13, p. 130506, 2011.

[24] C. Song, K. Xu, W. Liu, C.-p. Yang, S.-B. Zheng, H. Deng, Q. Xie, K. Huang, Q. Guo, L. Zhang, *et al.*, "10-qubit entanglement and parallel logic operations with a superconducting circuit," *Physical Review Letters*, vol. 119, no. 18, p. 180511, 2017.

[25] Y. Zhang, P. Li, Y. Jin, and Y. Choe, "A digital liquid state machine with biologically inspired learning and its application to speech recognition," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 11, pp. 2635–2649, 2015.