Temporal State Machines: Using Temporal Memory to Stitch Time-based Graph Computations

ADVAIT MADHAVAN, University of Maryland and National Institute of Standards and Technology
MATTHEW W. DANIELS and MARK D. STILES, National Institute of Standards and Technology

Race logic, an arrival-time-coded logic family, has demonstrated energy and performance improvements for applications ranging from dynamic programming to machine learning. However, the various *ad hoc* mappings of algorithms into hardware rely on researcher ingenuity and result in custom architectures that are difficult to systematize. We propose to associate race logic with the mathematical field of tropical algebra, enabling a more methodical approach toward building temporal circuits. This association between the mathematical primitives of tropical algebra and generalized race logic computations guides the design of temporally coded tropical circuits. It also serves as a framework for expressing high-level timing-based algorithms. This abstraction, when combined with temporal memory, allows for the systematic exploration of race logic–based temporal architectures by making it possible to partition feed-forward computations into stages and organize them into a state machine. We leverage analog memristor-based temporal memories to design such a state machine that operates purely on time-coded wavefronts. We implement a version of Dijkstra's algorithm to evaluate this temporal state machine. This demonstration shows the promise of expanding the expressibility of temporal computing to enable it to deliver significant energy and throughput advantages.

CCS Concepts: • **Hardware → Emerging architectures**; • **Computer systems organization → Data flow architectures**;

Additional Key Words and Phrases: Temporal computing, temporal state machines, graph algorithms

ACM Reference format:
Advait Madhavan, Matthew W. Daniels, and Mark D. Stiles. 2021. Temporal State Machines: Using Temporal Memory to Stitch Time-based Graph Computations. *J. Emerg. Technol. Comput. Syst.* 17, 3, Article 28 (May 2021), 27 pages.
https://doi.org/10.1145/3451214

1 INTRODUCTION

Energy efficiency is a key constraint when designing modern computers. The performance and efficiency of modern computers, which largely rely on Boolean encoding, can be attributed to developments across the computational stack from transistors through circuits, architectures, and other mid- to high-level abstractions. The recent stagnation of progress at the transistor level [32] is leading designers to make improvements at the lowest levels of the stack. These include

A. Madhavan and M. W. Daniels are contributed equally to this research.
Advait Madhavan acknowledges support under the Cooperative Research Agreement Award No. 70NANB14H209, through the University of Maryland.
Authors' addresses: A. Madhavan, University of Maryland and National Institute of Standards and Technology 100 Bureau Drive, Gaithersburg, MD 20899, United States; email: advait.madhavan@nist.gov; M. W. Daniels and M. D. Stiles, National Institute of Standards and Technology 100 Bureau Drive, Gaithersburg, MD 20899, United States; emails: {matthew.daniels, mark.stiles}@nist.gov.

1550-4832/2021/05-ART28 $15.00
https://doi.org/10.1145/3451214

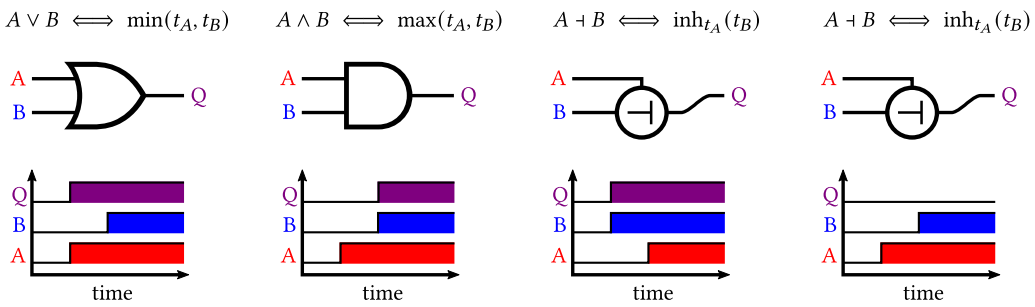

Fig. 1. Graphical examples of three race logic primitive operations Q = f(A, B). The top row indicates the correspondence between logical connectives and tropical functions; the bottom row contains waveforms of sample calculations. The inhibit gate in the two right-hand panels, which allows input B to pass only if it arrives before A, has no well-behaved analogue in Boolean logic.

re-imagining how data is encoded in physical states and introducing novel devices. The rationale is simple: Making the fundamental mathematical operations required for computation more efficient can have a cascading effect on the whole architecture. However, novel encoding schemes and devices come with new tradeoffs that differ from those of conventional Boolean computing schemes and which are not yet well understood.

In this article, we focus on an arrival-time encoding known as race logic [45]. Since digital transitions (edges) account for much of the energy consumption in traditional computation, race logic encodes multi-bit information in a single edge per wire. The arrival time t of this single edge is the value encoded by the signal. Encoding multiple bits on a single wire makes some operations very simple to implement. Standard AND and OR gates naturally implement the max and min functions; a unit delay element acts as an INCREMENT gate. A fourth logic gate, INHIBIT, allows its first (inhibiting) input to block the second input signal if the inhibiting signal arrives earlier. These operations are shown operating on examping inputs in Figure 1

The development of race-logic-based architectures has been largely *ad hoc*. Race logic was first developed to accelerate dynamic programming algorithms [45], and its application space has expanded to include machine learning [66] and sorting networks [51], demonstrating energy and performance advantages. Parallel development of logical frameworks [47, 60, 68], novel device technologies [46, 69], and fabricated chips [44] have contributed to a cross-stack effort to make this encoding scheme technologically viable. Here, we offer two important developments.

The first development is a systematized method of building computing architectures. An important step is to identify a suitable mathematical foundation that can express problems uniquely suited to a race logic approach. Formal logic, computation, and verification frameworks have been developed [60, 67, 68]. Continued progress requires identifying a mathematical algebra in which race logic *algorithms* and *state machines* are naturally expressed in the highly parallel dataflow contexts typical of temporal computing accelerators. We propose tropical algebra to be used in this context.

The second development is a compositional framework for programmatically linking low-level subroutines into higher-order functions. This development expands race logic beyond one-shot application-specific circuits. Recent work has started to explore several device concepts for efficiently reading and writing time-coded signals [46, 69]. The advantage of such memories is that they can directly interface with the temporal domain; read and write operations in such a memory can be performed without conversion to digital encoding.

The introduction of temporal memory technologies allows race logic to serve as an efficient computational fabric for two distinct but compatible advances. First, because memory breaks symmetries related to translations of the time coordinate, a temporal computer equipped with a memory is no longer subject to the invariance constraint on time-coded functions outlined in References [47, 60]. Lifting this restriction allows tropical algebra to serve as a coherent algebraic context for designing and interfacing race logic circuits. Second, memories allow us to reach beyond specialized one-shot temporal computations. Primitive race logic operations can be composed and iterated upon by saving outputs of one circuit and rerunning that circuit on the saved state, the temporal equivalent of a classical state machine. In this article, we develop the temporal state machine as an application-agnostic datapath for accelerating problems susceptible to temporal computing.

Our contributions are:

- A description of a temporal state machine that solves temporal problems in systematized parts, providing a clear computational abstraction for stitching larger computations out of primitive race logic elements.
- An exposition of tropical algebra as a mathematical framework for working with temporal vectors. We explain the mapping into tropical algebra from race logic and how it provides a convenient mathematical setting for working with temporal computations.
- Augmentations to conventional 1T1R (1 transistor, 1 resistor) arrays that make the crossbar architecture natively perform fundamental tropical operations. We use this, and other temporal operations, to create a more general feed-forward temporal computation unit.
- Demonstration and evaluation of a temporal state machine that uses Dijkstra's shortest path algorithm to find the minimal spanning tree on directed acyclic graphs.

The article is organized as follows: Section 2 briefly describes race logic and tropical algebra, showing the mapping between them. Based on that mapping, we describe circuit implementations of important tropical operations as the basic generators of higher order temporal functions. Section 3 introduces temporal state machines, explaining time-coded states and transition functions. We represent simple problems tropically and demonstrate how such a state machine can solve them in discrete steps. Section 4 presents a case study implementating Dijkstra's algorithm on a temporal state machine and proposes a purely temporal version of the algorithm. Performance and energy simulations follow in Section 5, followed by a comparison with previous work and discussion in Section 6.

2 TROPICAL ALGEBRA AND RACE LOGIC: MAPPING BETWEEN CIRCUITS AND SEMIRINGS

2.1 Race Logic and Temporal Computing

Computing with time traces back to two communities: one bio-inspired, and the other purely efficiency oriented. The biological interest in precise timing relationships between spikes grew after the seminal works by Thorpe on the processing speed of the human visual system [28, 65] and on spike timing dependent plasticity by Bi and Poo [7]. From then, temporal wavefront computation [21, 33, 56, 70] in the biological community expanded to the machine learning and neuromorphic computing communities [40, 53, 54]. References [8, 50, 55, 64, 73] show state-of-the-art performance and learning strategies in temporal neural networks, while the neuromorphic computing community in References [3, 17, 19, 25, 38, 53, 57] developed hardware to emulate precise timing relationships in spiking neural activity. More recently, precise timing-based codes in spiking neural networks perform a variety of applications such as graph processing [30, 36], median

filtering [71], image processing [71], and dynamic programming [1]. For several decades, the circuit community has independently been using time domain mixed signal analog techniques in Analog/Time to Digital Converters [52, 76], clock recovery circuits, phase and delay locked loops, phase detectors, and arbiters. With shrinking voltage levels and diminishing headroom, the temporal domain becomes attractive for analog processing. With the interest in emerging computing paradigms, this community has contributed temporal coded **complementary metal-oxide-semiconductor (CMOS)** only computational approaches [15, 20, 48, 58].

Race logic sits between the aforementioned approaches in that it uses biologically inspired wavefronts as the fundamental data structure, while using conventional digital CMOS circuits to compute. Race logic encodes information in the timing of rising digital edges and computes by manipulating delays between racing events. In the conventional Boolean domain, the electrical behavior of wires changing voltage from ground to V_{dd} is interpreted as changing from logic level 0 to logic level 1 at time t. In race logic, these wires are understood to encode each t as their value, since the rising edge arrives at t with respect to a temporal origin at $t = 0$. In some cases, a voltage edge can fail to appear on a wire within the allotted operational time of a race logic computation. In these cases, we assign the value temporal infinity, represented by the ∞ symbol.

We define race logic without memory elements as *pure* race logic, which accounts for most of the extant literature. We call race logic that uses dynamic memory elements *stateful* or *impure* race logic. Our goal here is to describe stateful race logic, but first we review issues that arise in pure race logic. The class of functions that can be implemented in pure race logic is constrained by physics [47, 60] through *causality* and *invariance*. The causal constraint, also called *non-prescience*, requires that the output of a race logic function be greater than or equal to at least one of the function's inputs. Any output must be caused by an input that arrives either earlier than or simultaneously with that output.

The invariance constraint arises because the circuit is indifferent to the choice of temporal origin. It is satisfied by race logic functions f for which $f(t_1 + \delta, t_2 + \delta, \ldots, t_N + \delta) = f(t_1, t_2, \ldots, t_N) + \delta$; all operations in pure race logic must obey this equality. Invariance need not apply to impure circuits, which contain a memory or state element: Such circuits perform differently at different times, depending on whether a memory element has been modified. From a programming perspective, a pure function always gives the same output when presented with the same input; an impure function is analogous to a subroutine that can access and modify global variables.

2.2 Tropical Algebra

Named in honor of Brazilian mathematician Imre Simon, tropical algebra treats the tropical semiring $\mathbb{T}$. In $\mathbb{T}$, the operations of addition and multiplication obey the familiar rules of commutativity, distributivity, and so on, but are replaced by different functions. *The tropical multiplicative operation is conventional addition, and the tropical additive operation is either min or max; the choice of additive operation distinguishes two isomorphic semirings.* Depending on the choice of min or max as the additive operation, the semiring is given by $\mathbb{T} = (\mathbb{R} \cup \{\infty\}, \oplus, \otimes)$ or $\mathbb{T} = (\mathbb{R} \cup \{-\infty\}, \oplus', \otimes)$; $\pm\infty$ are included to serve as additive identities.[1] These symbols, and others used in this article, are collected for reference in Table 1. That some of the generating operations of tropical algebra correspond directly to the primitive operations of race logic suggests that it is an ideal setting for the development of time-coded algorithms.[2]

[1] By contrast, the ring of real arithmetic is $(\mathbb{R}, +, \times)$.

[2] While tropical algebra is defined over the real numbers with infinity, a race logic circuit can practically represent only a finite discrete set of signal timings. Race logic and tropical algebra are therefore not isomorphic, *per se*. Note that the same

Table 1. List of Symbols Related to Race Logic and Tropical Algebra and Their Meanings

Symbol	Meaning	Description
∞	infinity	additive identity in tropical algebra; edge that never arrives in race logic
$\otimes$	add	multiplicative operation in tropical algebra; temporal delay in race logic
$\oplus$	min	additive operation in tropical algebra; first arrival in race logic
$\oplus'$	max	alternate additive operation in tropical algebra; last arrival in race logic
$\dashv$	inhibit	ramp function in tropical algebra; signal blocking in race logic
$=$	equivalence	expressing equality between two statements
$:=$	storage	storing a signal in memory
$:\cong$	normalized storage	storing a signal in memory by first performing a normalizing operation

Tropical algebra has found numerous applications in the computing literature particularly in a variety of graph algorithms, such as shortest path finding, graph matching and alignment, and minimal spanning trees. It is used as the basis of **GraphBLAS (Graph Basic Linear Algebra Subprograms)** [37]. In mathematics, it is being used to explore problems in combinatorial optimization [5], control theory, machine learning [80], symplectic geometry [6], and computational biology [77].

There are some fundamental similarities between tropical algebra and race logic. Both of them have an ∞ element. In race logic, it physically corresponds to a signal that never arrives, while in tropical algebra it corresponds to the additive identity, since

$$\alpha \oplus \infty = \min(\alpha, \infty) = \alpha. \tag{1}$$

Such an addition does not have an inverse, since there is no value of β in $\min(\alpha, \beta)$ that would give ∞. The non-invertibility of addition means that this algebra is a semiring and fundamentally winner-take-all in nature. Every time the additive operation is performed, the smallest number (the first arriving signal in race logic) "wins" and propagates further through the computation.[3] The multiplicative identity in tropical algebra is zero, rather than one, since $\alpha \otimes 0 = \alpha + 0 = \alpha$.

2.3 Graph Problems in Tropical Algebra

Tropical algebra can be especially useful for graph analytics, where it provides a simple mathematical language for graph traversal. A fundamental concept in graph traversal is the graph's weighted adjacency matrix A. Figures 2(a) and (b) show a directed graph and its weighted adjacency matrix, respectively. The ith column of the weighted adjacency matrix represents the distances of the

is true of a traditional computer with respect to conventional real arithmetic. However, just as traditional computers can operate over a large enough subring of the reals to produce useful calculations, there exists an embedding of min-based race logic as a subsemiring of tropical algebra. Regardless of whether we work in a subset of natural numbers (clocked race logic) or the reals (analog race logic), the fact that this mapping is well-behaved ensures that tropical algebra is a useful mathematical landscape for understanding race logic operations.

[3]If we had chosen $\oplus'$ instead of $\oplus$, then the additive identity would be $-\infty$; though, we generally prefer the min-plus version of tropical algebra. In pure race logic, ∞ corresponds to an edge that never arrived, whereas $-\infty$ would correspond to an edge that had always been present—not to be confused with an edge that arrived at $t = 0$. No nontrivial function in pure race logic can output $-\infty$ due to the causality constraint, so the min-plus algebra has considerably more practical utility in race logic.

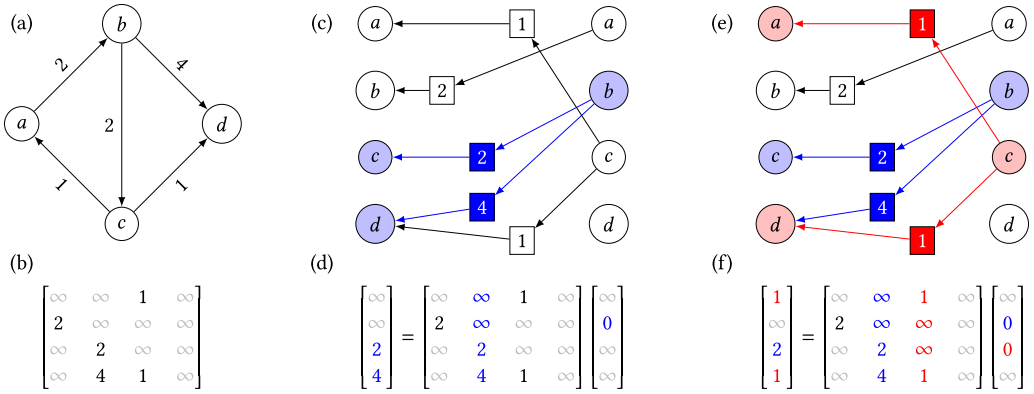

Fig. 2. Tropical matrices for graph exploration: (a) shows an example directed graph; (b) shows the equivalent weighted adjacency matrix. Panel (c) shows the propagation of a signal originating at node b through a delay network corresponding to the edges of the example graph. Panel (d) shows the tropical vector-matrix multiplication corresponding to panel (c). Panels (e) and (f) repeat these representations for the case where signals are injected at both b and c.

outward connections from node i to all other nodes in the graph, so A_{ji} is the weight for the edge $i \to j$. Where there is no edge to node i from j, we assign the value $A_{ji} = \infty$.

The usefulness of tropical algebra for graph traversal is seen when using A in a tropical vector-matrix multiplication. Tropical **vector-matrix multiplication (VMM)** proceeds like conventional VMM, but with $(\oplus, \otimes)$ instead of $(+, \times)$. As shown in Figure 2, each vector element is scaled (tropical multiplication) before they are all accumulated (tropical addition). Extracting any single column from a matrix can be done by multiplying a one-hot vector, as shown in Figure 2. The tropical one-hot vector has a single zero element with all other entries set to ∞; from Section 2.2. During scaling, the columns of the adjacency matrix that correspond to the infinities of the one-hot vector get scaled to infinity (tropically multiplied by ∞) while the remaining column, scaled by the multiplicative identity 0, is the output. The values stored in the output vector represent the distances from the one hot node in the input vector. This operation represents a search from the node in question (decided by the one-hot vector) to all the connected nodes in the graph and reports the distances along all edges of this parallel search.

Using a "two-hot" vector for input, as shown in Figure 2(d) outputs a tropical linear combination of two vectors, corresponding to the "hot" columns of the adjacency matrix. The accumulation phase of the tropical VMM is nontrivial; the $\oplus$ operation selects the smallest computed distance to each node for the output. The tropical VMM reports the shortest distance to each node in the graph after a single edge traversal from *either* of the initial nodes specified by the two-hot vector. Both steps—the exploration of a node's neighbors and the elementwise minimum of possible parent nodes associated with an output—are performed in parallel by a single matrix operation.

Representing a collective hop through the graph as a single matrix operation allows a series of matrix operations to represent extended graph traversal. The shortest traversed distance to each node in a graph from an initial node x is

$$y = x \oplus (x \otimes A) \oplus (x \otimes A \otimes A) \oplus (x \otimes A \otimes A \otimes A) \oplus \cdots . \tag{2}$$

The first term represents all single-hop shortest paths starting out from x, while the second term accounts for all the two-hop shortest paths, and so on. Hence, the tropical summation across all the terms in y allows it to encode the shortest distances between the input node as specified by x,

independent of the number of hops. Performing N such hops and calculating the minimum distance across all of them is the key operation in various dynamic-programming-based shortest path algorithms. This process makes tropical algebra the natural semiring for Dijkstra's algorithm [49]. We use these ideas to implement Dijkstra's single-source shortest path algorithm in a stateful race logic system in Section 4.

2.4 Circuits for Tropical Linear Algebra

Since tropically linear functions $\bigoplus_j (a_j \otimes t_j)$ with the a_j values constant satisfy the invariance condition, tropical linear transformations may be carried out in pure race logic. In Section 2.1, we describe how single rising edges can be used to encode information in their arrival time. Interpreting the edges as tropical scalars, we can see how OR gates and delay elements are modeled by tropical addition and multiplication. This understanding can also be extended to tropical vectors. Section 2.3 describes how tropical vectors can be interpreted as distance vectors in graph operations. These distance vectors can be interpreted temporally as a wavefront or a volley of edges measured with respect to a temporal origin. Other researchers have proposed using such vectors as the primary data structure underlying temporal computations [60, 64].

Just like conventional vectors, tropical vectors have a normalization rule, but it is somewhat unusual. Normalization proceeds by subtracting the minimum element of a vector from all elements, ensuring that at least one element of the normalized vector is equal to zero.[4] It is common to regard a tropical vector as equivalent to its normalized version, implying that it only encodes information about the shape of its temporal wavefront, and not about an arbitrarily chosen temporal origin. To accept this equivalence is to work in the *projective* tropical vector space, and we refer to tropical vectors as being projectively equivalent if their normalized versions are elementwise equal. Frequent renormalization has the advantage of mitigating overflow, and simple circuit modifications required to implement it are discussed in Section 3.1. We use and account for normalization algorithmically in Section 4.1, allowing us to encode information in a principled way that would nominally extend beyond our dynamic range.

Once a wavefront of rising voltage edges is interpreted as a tropical vector, the techniques shown in Figure 3 can be used to implement tropical vector operations. Panel (a) shows the vectorized version of the tropical dot product operation. Abstracting out for now implementation details of the individual delay elements, the column delays each line of the incoming wavefront by a different amount. This implements tropical multiplication by constants and can be seen as superimposing the delay wavefront onto the incoming wavefront. The outputs of such a circuit are then connected to the inputs of a pre-charge-based pullup with an OR-type pulldown network followed by an inverter. The circuit operation is divided into two phases, the pre-charge phase followed by the evaluation phase. In the pre-charge phase, the PMOS transistor has its input connected to ground, causing the critical node to be pulled-up (connected to V_{dd}). When the pre-charge phase ends, the PMOS transistor is turned off, which maintains the potential at the critical node at V_{dd}. During the evaluation phase, the first arriving rising edge at the input of one of the NMOS transistors causes the critical node to discharge to ground, hence being pulled-down to a potential of zero volts. This behaves as a first-arrival detection circuit that outputs a rising edge at the minimum of the input arrival times, performing the min operation. It implements tropical vector addition. Combining the delay (multiplication) with the min (summation), we get the tropical dot product operation. By replicating this behavior across multiple stored vectors, as in panel (b), we get the tropical VMM operation, where the input vector tropically multiplies a matrix.

[4] In the max-plus tropical semiring, the vector norm would be the maximum element of the vector. This vector magnitude operation is sometimes called the L^∞ norm.

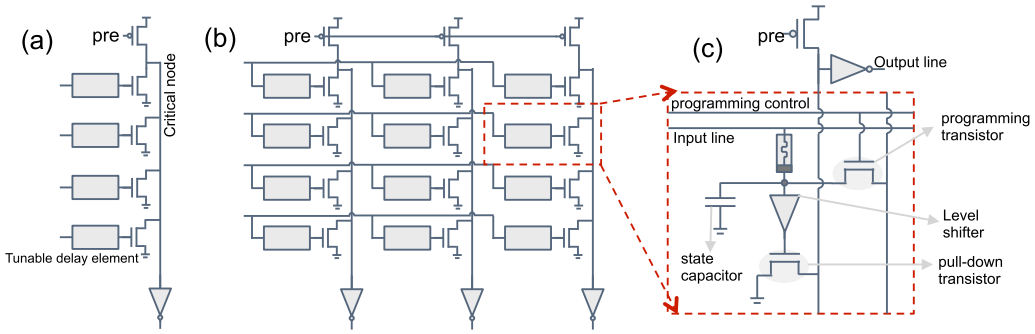

Fig. 3. Construction of race logic circuits for tropical algebra: Panel (a) shows a composite circuit for the tropical dot product operation. A simple array of delay elements takes an incoming wavefront and delays its elements by the values stored in each of the delay elements. This represents the tropical element-wise multiplication by constant operation. The output is then connected to a *p*-type, metal-oxide semiconductor (PMOS) pre-charge pullup coupled with a NOR-style pulldown network that behaves like a first arrival detector and performs the tropical addition operation. Panel (b) combines multiple elements of panel (a) and scales this up to a 2D array such that it performs tropical vector matrix multiplication, the critical operation for graph traversal, as described in Section 2.3. Panel (c) shows a detailed circuit implementation of the tropical VMM cell. Each cell consists of two transistors, one for programming and the other for operation, and a level shifter [46]. In the programming mode, the array is used like a conventional 1T1R array and the memristors are written to the appropriate resistance values. In the operation mode, the programming transistor is turned off, while the gate capacitor (shown in figure) is charged through the memristor. The level shifter is used to make sure that the discharge time constant is determined by the memristor charging process and not the pulldown of the transistor, by applying full swing inputs to the pulldown transistor.

To be specific, we consider versions of the tunable delay elements described in the previous paragraph that are based on memristor or **resistive random access memory (ReRAM)** technology. CMOS and spintronic versions of such delay elements are discussed in Section 3.1. In the tropical VMM such a device is used as a programmable resistor with a known capacitance to generate an *RC* delay [46]. The details of these tropical vector algebra cells are shown in Figure 3(c). The main element of this circuit is a 2T1R array composed of a pulldown transistor and a programming transistor. During the programming phase, the programming transistor coupled with the programming lines can be used to apply the necessary read and write voltages across the memristor, thus changing the resistance and therefore *RC* delay time stored in the device.[5] During the operation of the circuit, the programming transistor is turned off to decouple the programming lines from the active circuitry. In the pre-charge phase, the output lines are pulled up to V_{dd} through the pullup transistor. In the evaluation phase, the input lines receive temporally coded rising edges that charge up the gate capacitors, as shown in Figure 3. This causes the pulldown transistor to be turned on at the time proportional to input arrival times plus the *RC* time constant of the coupled memristor-capacitor in each cell, faithfully performing the tropical VMM operation.

The largest read voltage that can applied across the device without disturbing the state of the device is approximately 600 mV. In a 180 nm process, this value is only a few 100 mV above the transistor threshold voltage and would cause a slow and delayed leak. This leak allows multiple inputs to affect the pulldown simultaneously, influencing the functional correctness of the circuit. We propose two solutions to this problem. Figure 3(c) shows a level shifter added between the

[5]The details of the programming and precision for these devices is discussed in Section 6.

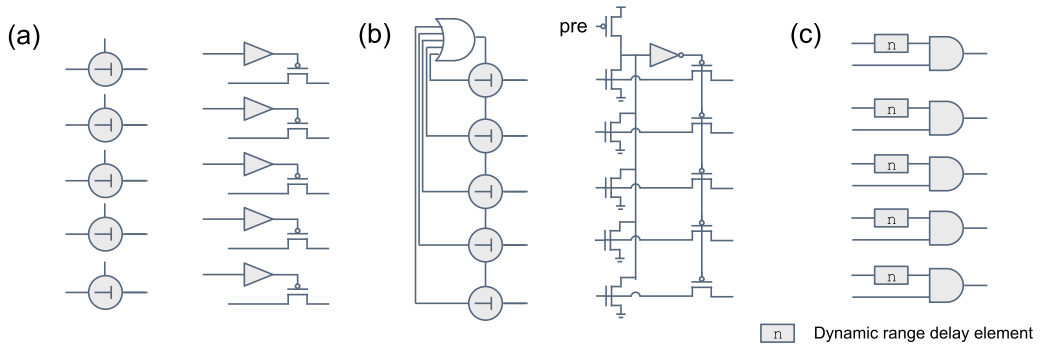

Fig. 4. Tropically nonlinear race logic functions: Panel (a) shows the conceptual and circuit diagram for an element-wise inhibit operator. The inhibiting input is buffered before being fed into the gate terminal of a PMOS. As the inhibiting input turns high, the PMOS turns off inhibiting the secondary input. Panel (b) shows the argmin operation that takes an input vector and returns a one-hot vector at the location of the element with the minimum value. This is done by taking the first arrival signal and inhibiting everything else but that signal. Panel (c) shows a binarizer. An input wavefront is maxed with the all n wavefront. This takes all values to this max value, except ∞, which remains as is, performing binarization.

memristor and the pulldown transistor, the full swing of which causes the pulldown transistor to work much faster. In an alternate approach (not shown here), a medium V_{th} device is used for the pulldown. Such devices ensure a small footprint as well as correct operation, provided the fabrication process allows them.

In addition to tropical VMM based linear transformations, other primitive vector operations are crucial in many real applications. Elementwise min and max can be performed with arrays of OR and AND gates, respectively. Vectors can also be reduced to scalars by computing min or max amongst all elements using multi-input OR and AND gates.

2.5 Circuits for Nonlinear Tropical Functions

Apart from circuits that allow race logic to implement tropical linear algebra, additional built-in functions, such as elementwise inhibit, argmin, and binarization, are required to perform a wider variety of tropical computations. Elementwise inhibit, shown in Figure 4(a), is particularly powerful, as it allows us to implement piecewise functions. Its technical operation follows directly from the scalar inhibit operation discussed in Section 2.1.

The argmin function, shown in Figure 4(b), converts its vector input to a tropical one-hot vector that labels a minimal input component. An OR gate is used to select a first arriving signal that then inhibits every vector component. Only one first arriving edge survives its self-inhibition; no other signals in the wavefront are allowed to pass, effectively sending these other values to infinity. The resulting vector is projectively equivalent to a tropical one-hot and achieves the canonical form with a single zero among infinities after normalization.[6]

The binarization operation, shown in Figure 4(c), is similar; it converts all finite components to 0 while preserving infinite components at ∞. This operation utilizes a pre-stored vector that has the maximum finite (non-∞) value $t_{\max}$ of the computational dynamic range on each component.

[6]A variety of conventions could be taken for the case where more than one signal arrives at the same, earliest time. Note that such a multi-hot vector can be generated by the circuit shown in Figure 4(b). When such a situation occurs, a sorted delay vector can be used to select one of the hot input elements and convert the vector to a one-hot vector.

We define binarize($\vec{x}$) = $t_{\max} \oplus' \vec{x}$. Computing the elementwise max of such a vector with any incoming vector, values that are ∞ remain so while the other values are converted to the maximal finite input value. Normalizing the result via projective storage saves a many-hot vector labeling the finite components of the original input.

3 TEMPORAL STATE MACHINES

The *finite state machine* or *finite state automaton* is a central concept in computing and lies at the heart of most modern computing systems. Such a machine is in one of some finite set of states S at any particular instant in time; inputs $x \in \Sigma$ to the machine both produce outputs $y \in \Gamma$ and induce transitions between these internal states. A *state transition function* $\delta : S \times \Sigma \to S$ determines the next state based on the current state and current input, and an *output function* $\omega : S \times \Sigma \to \Gamma$ gives the output based on the state and inputs of the machine.[7]

The presence of state means that there is not a one-to-one correspondence between input and output of the machine. In the language developed above, a state machine is an impure function even though δ and ω are pure functions. The finite state machine provides a template for composing pure race logic functions across stateful interfaces to create more flexible processing units. The temporal state machines we introduce below fit into this mathematical framework. They differ from conventional automata in that the signals use temporal rather than Boolean encoding. State is made possible by temporal memories that use wavefronts as their primary data structure. They freeze temporal data by coupling pulse duration to device properties such as resistance. Together with the pure race logic primitives described in previous sections, temporal memories enable end-to-end temporal encoding in finite state automata.

Designing such a machine requires addressing several problems intrinsic to temporal logic and memory primitives. We start this section with some brief background on temporal memories. Then, we describe the impure tropical multiplication of two signals in race logic as an example of composing pure race logic across stateful interfaces to break through the invariance restriction. Finally, we return to the general state machine formulation and argue for the extensibility of our simple example to more complex systems.

3.1 Temporal Memory

Temporal memories natively operate in the time domain. They operate on wavefronts of rising edges rather than on Boolean values. Such memories can be implemented in CMOS [15, 22, 58] and with emerging technologies such as memristors (as shown in Figure 5) and magnetic race tracks [69]. In CMOS, SRAM-based volatile digital storage is predominantly used to select between various delays, controlled either by varying capacitor sizes or the number of inverting stages. Writing into these memories is accomplished by time-to-digital converters or Vernier delay lines. By contrast, the device physics of emerging nanodevice technologies provides a direct coupling between physical time and an analog device property. Regardless of delay element implementation, memory cells are arranged in a crossbar. For the read operation, a single rising edge represented by the tropical one-hot vector is applied to the input address line of the memory, creating a wavefront at the output data line. For a write operation, the column of the crossbar where the memory has to be stored is activated and an incoming wavefront is captured. Here, we adopt memristor-based memory for concreteness. However, the algorithmic concepts presented in this article are independent of that choice.

[7] This specification of a state machine is called a Mealy machine; if ω depends only on the state and not the current input, then it is called a Moore machine. The two models are equally powerful in principle.

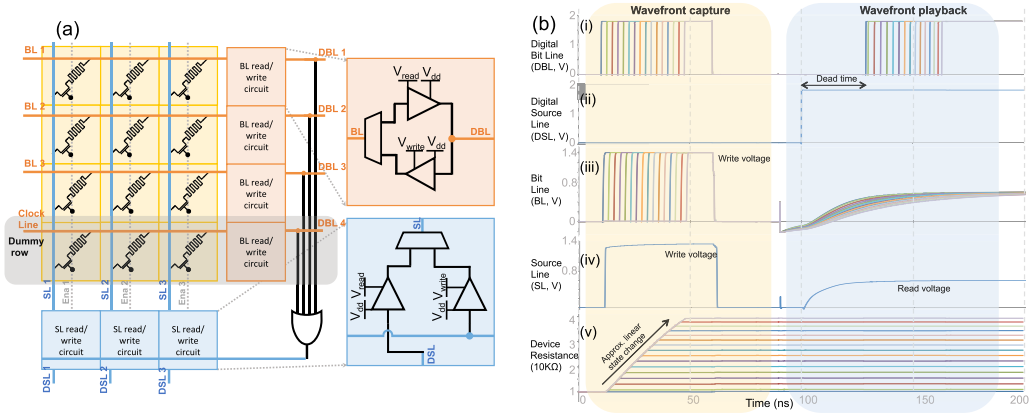

Fig. 5. Memristive wavefront memory: Panel (a) shows a 4×4 memristive temporal memory, complete with read and write peripheral circuits as described in Reference [46]. Note that bit-line-4 has been replaced by a dummy line where the resistance values are fixed. The time constant of this line is governed by the parasitics of the circuit and determines the temporal origin of the outgoing wavefront. Panel (b) shows the functioning of a 16×16, 4-bit, temporal memory as simulated in our 180 nm process. Strip (i) shows the capture and playback of a linearly varying digital wavefront, with each color representing one of the 16 lines involved. These edges have been collapsed into a single strip for clarity. Note that small timing mismatches cause small changes in the shape of the wavefront that is played back. Strip (ii) shows in the digital read input applied to a captured column. Strips (iii, iv) show the source lines and bit lines internal to the memory. These lines operate at different voltages that are shifted to V_{dd} with level shifters, as shown in panel (a). Strip (v) shows the almost linear state change of the memristors as given by the memristor model in Reference [14]. Careful inspection reveals a slight convexity due to higher order terms in the exponential dependence.

The temporal information encoding in the devices varies with technology. For memristors, it is encoded in the *RC* charging time constants determined by the resistance R of the memristor and the row capacitance C, leading to a linear relationship between timing and resistance. Utilizing a 1T1R-like structure, the shared row capacitances are the output capacitances that have to be charged. In the write operation, the different arrival times of the edges lead to different durations of high voltage across the memristors creating resistance changes proportional to the durations, correctly encoding the shape of an incoming wavefront. This has the advantages of being non-volatile and compact, having analog tunability, and requiring fewer transitions than their CMOS counterparts, hence being potentially more energy-efficient.

Temporal memories have a disadvantage over conventional memories based on registers. In conventional memories, a single clock tick performs two functions. It captures the next state information from the calculation performed in the previous cycle and initiates the next cycle's computation. Combinational logic is "stitched" together by register interfaces. On the other hand, temporal wavefront playback and capture use the same address and data lines, shown in Figure 5, and cannot be used at the same time. This feature of temporal memory does not exist in conventional memories, which can be used both upstream and downstream for the same operation.

Temporal memories based on memristors have some disadvantages compared to temporal memories based on CMOS. Because they are analog,[8] they possess limited dynamic range. They also

[8]The computing scheme discussed here can be either analog or digital. Though our evaluation (Section 5) is done assuming analog behavior, noise, and other non-idealities determine the practical information capacity. We discuss this issue in Section 6.

have dead time, as shown in 5(b), that results from the charging of the parasitics of the array, which—with growing array size—can become comparable to the delays stored. As measured from the temporal origin of the calculation, the dead times introduce artificial delays in each component resulting in incorrectly encoded values at a memory write input. To deal with this issue, we introduce an extra line, which we call the *clock line*, that always has the minimum R_{on} value for the resistor. This line serves as a temporal reference to the origin and hence behaves like a clock. This ensures that the parasitics of the lines are accounted for and only the relative changes in the resistance values are translated to the output wavefront. Such a reference also plays an important role in normalization.

The dynamic range/precision of memories discussed is determined by the changes in the stored resistances that can be distinguished. Even optimistically, the range is limited to six to seven bits with present technologies. Given this constraint, we focus on storage of normalized tropical vectors. In Section 2.2, we describe how normalization consists of subtracting from each component the minimal value of all components. Operationally, this is performed by grounding the clock line of our vector storage circuit in Figure 5. The clock line is used to indicate the $t = 0$ time, but in its absence $t = 0$ is established by the first arriving edge. Grounding the clock line effectively subtracts the value of the first-arriving edge from the entire vector, giving the normalization operation we described in Section 2.2. The algorithms explored in Section 4, are specifically designed to be insensitive to this origin shift. Algorithms can also store the normalization constant for later re-construction of the unnormalized wavefront.

3.2 Invariance and Temporal Addition

In Section 2.1, we describe how invariance restricts pure (stateless) race logic to tropically linear functions. Therefore, pure race logic cannot tropically multiply two temporal signals. Static delay elements can increment the value of a temporal signal by some fixed amount, but the raw addition of two time codes $t_1 + t_2$ is physically forbidden by time-translational symmetry.[9] The introduction of memory breaks this symmetry in stateful race logic.

With temporal memory, tropical multiplication of two wavefronts breaks the operation into two phases, as shown in Figure 6(a,b). Panel (a) shows the first phase, storing the incoming wavefront in a local temporal memory using wavefront capture circuits. In the second phase, shown in panel (b), this stored vector is temporally added to an incoming wavefront. Commutativity ensures that the order of storage and playback does not matter. Though the state transition and output functions within each phase are pure race logic functions, state breaks invariance across the phase boundaries. Using memory for tropical multiplication allows the construction of tropical multinomial functions of arbitrary order.

3.3 A Sample Temporal State Machine

The invariant race logic circuits and temporal wavefront memory described above are sufficient to build a simple temporal state machine, as shown in Figure 6(a). It consists of three banks of temporal memory, which can receive address inputs from external sources as well as data inputs from the output of the machine. The data outputs of the wavefront memory are multiplexed into the computation unit. This unit contains the invariant race logic functions from Section 2.1 and

[9]Invariance (Section 2.1) requires that if two signals t_A and t_B are both shifted by a constant time δ, then the output of a function of those signals must also be shifted by the same amount, i.e., $f(t_A + \delta, t_B + \delta) = \delta + f(t_A, t_B)$. Addition, $f(t_A, t_B) = t_A + t_B$, violates this requirement, because shifting the inputs gives $t_A + t_B + 2\delta$ at the output. Addition is not temporally invariant prohibiting the addition of two temporal signals in pure race logic.

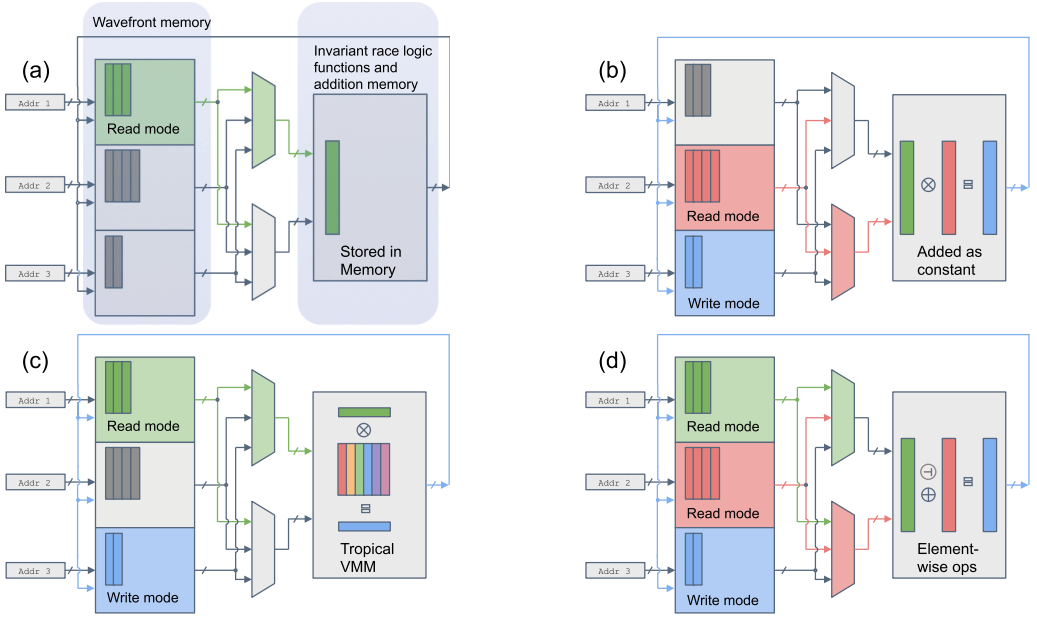

Fig. 6. State machine operations: (a) The first phase of tropical multiplication. The state machine is partitioned into two main units: the temporal wavefront memory and the arithmetic unit. Multiplexers and the read/write modes of the memory allow operations to be performed sequentially. Depending on the operations, individual memory units can behave as either upstream or downstream memories. Storage of the incoming wavefront is a one-argument operation; the vector is stored in the additive memory bank. (b) The second phase of tropical multiplication. The incoming wavefront is delayed component-wise (tropically multiplies) by the stored wavefront. (c) The tropical VMM operation. (d) Other element-wise operations that can be performed in a temporal state machine. All operations, aside from the first phase of the tropical multiplication, store an output back in the temporal memory. The element-wise operations are two-argument operations and involve all three memories: The read memories are the upstream memories, while the write memory is the downstream memory.

temporal memory unit for tropical VMM, as described in Section 2.4. This structure allows for a maximum of two-operand operations to be executed at once.

ALGORITHM 1: Pseudocode for procedural computation of Equation (3)

Input: temporal vectors $\vec{b}$, $\vec{c}$, $\vec{d}$, and $\vec{e}$

$\vec{c}' := \vec{d} \otimes \vec{e}$; `// temporal vector addition (requires two phases), Figures 6(a,b)`

$\vec{b}' := \vec{c} \dashv \vec{c}'$; `// elementwise inhibit, Figure 6(d)`

$\vec{a} := \vec{b} \oplus \vec{b}'$; `// elementwise min, Figure 6(d)`

return $\vec{a}$;

This state machine partitions calculations into phases to calculate arbitrary expressions such as

$$\vec{a} = \vec{b} \oplus (\vec{c} \dashv (\vec{d} \otimes \vec{e})). \tag{3}$$

Each phase is implemented serially on the state machine in Figure 6. Breaking the computation into discrete read-compute-store transitions of a state machine allows us to represent the computation with a procedural algorithm, Algorithm 1.

Using the regular order of arithmetic, we perform tropical multiplication first. Vectors $\vec{d}$ and $\vec{e}$ reside in memories one and two. The $\otimes$ operation is shown in Figure 6(a,b). The first phase initiates the computation by selecting the memory in the computation unit and applying a one-hot vector at the input of wavefront memory 1. The memory places the vector $\vec{d}$ on the output data bus, which then passes it to the accumulator of the computation unit. Then, memory 3 is set up to receive the output of the operation while being activated in write mode, shown in Figure 6(b). A one-hot vector is applied to the input of memory 2, playing the wavefront through the stored vector, and storing the resulting output in memory 3. This storage operation is indicated by the assignment operator := in the pseudocode. Tropical vector-matrix multiplication is a similar one-input operation and can be performed in a similar way, as shown in Figure 6(c).

Both one- and two-operand operations can be performed in a single state machine. Two-operand operations, such as elementwise inhibit and tropical vector addition, proceed similarly to one-operand operations. Synchronized one-hot vectors are presented to the address input that triggers output wavefronts. These wavefronts enter the computational unit where circuits for the requested operations are multiplexed in, and the output is written to wavefront memory 3, as shown in Figure 6(d).

3.4 A Nontrivial Example: DNA Alignment

DNA alignment using a temporal instantiation of the Needleman-Wunsch algorithm was one of the first applications of race logic [44, 45]. In that work, the alignment matrix of the Needleman-Wunsch algorithm is physically laid out as a planar graph, and pure race logic operations define the scoring information at each node. Though the implementation in Reference [44] is extremely fast and energy efficient, it requires a dedicated ASIC. Here, we sketch how Needleman-Wunsch might be implemented with a temporal state machine.

The Needleman-Wunsch algorithm finds the shortest path through a dynamically constructed score matrix. Each element of the score matrix M_{ij} is constructed recursively as $M_{ij} = \min\{M_{i,j-1} + \sigma, M_{i-1,j} + \sigma, M_{i-1,j-1} + \boldsymbol{m(1 - \delta xi, yj)}\}$, where σ is the cost of a genetic insertion or deletion (an "indel") and m is the cost of a single gene mutation.[10] The Kronecker delta function breaks the causality condition and so cannot be implemented in pure race logic.

To compute the Kronecker delta, we encode the set of four possible genes $\{\mathtt{G}, \mathtt{A}, \mathtt{T}, \mathtt{C}\}$ as temporal values $\{0, 1, 2, 3\}$. We then use the coincidence function [47, 60] to determine equality of the temporally encoded gene values. Tropically the coincidence function is described as $\delta(t_1, t_2) = (t_1 \oplus t_2) \dashv (t_1 \oplus' t_2)$, which is equal to the inputs when they are the same,[11] and ∞ otherwise [47]. The coincidence function could be made a primitive operation of the state machine or could be accomplished with multiple state transitions using $\oplus$, $\oplus'$, and $\dashv$; we assume the former. Binarization followed by projective storage of $\delta(x_i, y_j)$ would save zero (tropical one) to memory when $x_i = y_j$ and ∞ (tropical zero) otherwise, resulting in a many-hot vector that indexes genewise equality.

[10] The Kronecker delta δ_{ij} is defined as one when $i = j$ and zero otherwise.

[11] The simple version presented in the text applies to only an idealized coincidence: the exact point where $t_1 = t_2$. In practice [47, 60], a nonzero coincidence window can be introduced via a tolerance ϵ, by computing $[\epsilon \otimes (t_1 \oplus t_2)] \dashv (t_1 \oplus' t_2)$.

ALGORITHM 2: Pseudocode for Needleman-Wunsch (forward pass only; computes optimal alignment cost)

Input: gene sequences $\vec{x}, \vec{y} \in \{0,1,2,3\}^n$, indel cost σ, mismatch cost m
$\vec{\mu}^{(0)} := [0]$;
$\vec{\mu}^{(1)} := [\sigma, \sigma]$;
// Upper-left triangular part [$\dim(\vec{\mu}^{(k)})$ increasing]:
for $k \leftarrow 2$ **to** n **do**
  $\vec{c}' := \delta\,(\vec{x}_{1,\ldots,k-1}, \vec{y}_{k-1,\ldots,1})$; // mismatches $\to \infty$, matches $\to \{0,1,2,3\}$
  $\vec{c} :\cong \text{binarize}(\vec{c}')$; // mismatches $\rightsquigarrow \infty$, matches $\rightsquigarrow 0$
  $\vec{a} := \sigma \otimes \vec{\mu}^{(k-1)}$; // apply insertion/deletion (indel) cost σ
  $\vec{b} := (m \oplus \vec{c}) \otimes \vec{\mu}^{(k-2)}$; // apply mutation cost m for mismatches
  $\vec{r} := \vec{a}_{0,\ldots,k-2} \oplus \vec{b} \oplus \vec{a}_{1,\ldots,k-1}$; // find least-cost local path (Equation (4))
  $\vec{\mu}^{(k)} := [a_0, \vec{r}, a_{k-1}]$; // append boundary conditions
end
// Lower-right triangular part [$\dim(\vec{\mu}^{(k)})$ decreasing]:
for $k \leftarrow n+1$ **to** $2n$ **do**
  $\vec{c}' := \delta\,(\vec{x}_{k-n,\ldots,n}, \vec{y}_{n,\ldots,k-n})$; // mismatches $\to \infty$, matches $\to \{0,1,2,3\}$
  $\vec{c} :\cong \text{binarize}(\vec{c}')$; // mismatches $\rightsquigarrow \infty$, matches $\rightsquigarrow 0$
  $\vec{a} := \sigma \otimes \vec{\mu}^{(k-1)}$; // apply insertion/deletions (indel) cost σ
  $\vec{b} := (m \oplus \vec{c}) \otimes \vec{\mu}^{(k-2)}$; // apply mutation cost m for mismatches
  $\vec{\mu}^{(k)} := \vec{a}_{0,\ldots,2n-k} \oplus \vec{b} \oplus \vec{a}_{1,\ldots,2n-k+1}$; // find least-cost local path (Equation (4))
end
return $\vec{\mu}^{(2n)}$; // this is actually just a scalar: lowest possible alignment cost

To frame the Needleman-Wunch algorithm as a tropical vector problem, we exploit the independence of the skew-diagonals [43]. We define $\vec{\mu}^{(k)}$ as the kth skew-diagonal vector of M, so $\vec{\mu}^{(0)} = [M_{00}]$, $\vec{\mu}^{(1)} = [M_{10}, M_{01}]$, and so on. The first and last elements of $\vec{\mu}^{(k)}$ are $k\sigma$ by construction for $k \leq n$, that is, until we hit the main skew diagonal. The defining equation for M_{ij} is then given through $\vec{\mu}^{(k)}$ by

$$\mu_j^{(k)} = \left(\sigma \otimes \mu_j^{(k-1)}\right) \oplus \left(\left[m \oplus \delta_{x_j, y_{k-j}}\right] \otimes \mu_j^{(k-2)}\right) \oplus \left(\sigma \otimes \mu_{j+1}^{(k-1)}\right). \tag{4}$$

The vectorized computation of this recursion relation is presented programmatically in Algorithm 2. The right-hand side of each assignment is a pure race logic computation; the left-hand side represents a register address. As in Algorithm 1, the assignment operator $\vec{x} := \vec{y}$ indicates storage of $\vec{y}$ to a temporal memory register represented by $\vec{x}$. The projective storage operator $\vec{x} :\cong \vec{y}$ assigns the tropical normalization $\vec{y} - \min \vec{y}$ to the vector register $\vec{x}$.

The interpreter required here is more complex than in Algorithm 1. Though we could implement the **for**-loops tropically by assigning $k := 1 \otimes k$ and monitoring $n \dashv k$ and $2n \dashv k$, we are not aware of a way to elegantly perform subarray extraction using temporal signals as indices. We therefore imagine that k, as well as the array slicing operations, are managed digitally by the interpreter.

4 CASE STUDY: DIJKSTRA'S ALGORITHM IMPLEMENTED IN A TEMPORAL STATE MACHINE

In Section 3, we demonstrate a simple model state machine, but it is too simple to utilize the graph traversal logic of tropical linear algebra that we describe in Section 2.3. Though the

Needleman-Wunsch machine in Section 3.4 performs graph traversal, it is restricted to a known, uniform progression through a highly regular planar graph. From the discussion of Section 2.3, we know that general graph traversal should be accessible to a tropical state machine. Here, we discuss an implementation of Dijkstra's algorithm in a temporal state machine. We see that the core neighbor-search operation of Dijkstra's algorithm is naturally parallelized by the tropical VMM, leading to very high throughput in terms of graph edges traversed per unit time, and that the inhibit operation together with projective storage allow the embedding of important Boolean logic structures within the temporal framework.

4.1 Dijkstra's Algorithm in Race Logic

We assume that the reader is familiar with the classical implementation of Dijkstra's algorithm. In Algorithm 3, we map the operations of Dijkstra's algorithm into race logic, with each step a single transition of a temporal state machine. Two trivial modifications simplify the race logic implementation. First, instead of tracking the known distances to each node, we mask out the distances of visited nodes with the value ∞. This vector of distances to unvisited nodes is $\vec{d}$ in the algorithm listing, and a tropically binarized record of which nodes have been visited is recorded in a vector $\vec{v}$. Second, instead of storing a parent vector directly, we define a parent matrix $\hat{P}$ as a collection of tropical column vectors where a finite entry P_{ij} holds the distance from node i to node j along the current optimal path to j from the source node s. We assume that the memristors in the VMM are already programmed to their correct values, meaning that the graph is already stored in the arithmetic unit.

There are several apparent differences in how operations of the algorithm are performed in this (tropical) linear algebra engine compared to a traditional programming language. There are, loosely speaking, two "modes" in which we use tropical vectors. First, there are true temporal wavefronts, such as $\vec{e}$ and $\vec{d}$, that represent variable distances measured throughout the graph. These flow through the data path of the algorithm. Second, there are indicator wavefronts, such as $\vec{v}$ and $\vec{d}^*$, with elements restricted to 0 or ∞. These are used along the control paths of the algorithm to perform element lookup from data-carrying temporal wavefronts, modification of tropically binary records such as $\vec{v}$, and for index selection of the parent matrix. Projective storage plays a key role in these processes via binarization of one-hot vectors. Sometimes, quantities like $\vec{n}$ can play either role depending on context.

There are two primary constraints on this algorithm's application. First, because directed edge weights are encoded as temporal delays, negative edge weights are physically forbidden. Second, temporal vectors are limited to a finite dynamic range and resolution constrained by the technology in which they are implemented, and consecutive tropical multiplication could lead to dynamic range issues. To mitigate this dynamic range issue, we arrange the computation such that no more than one successive tropical multiplication occurs along a single datapath per state machine transition. Normalization of $\vec{u}$ at the end of each cycle shrinks the dynamic range as much as possible between VMMs.

The algorithm initializes by setting the vector $\vec{d}$ of known distances to unvisited nodes to a tropical one-hot $\mathbf{0}_s$ labeling the source node s. The vector $\vec{v}$ labeling visited nodes, as well as the parent matrix $\hat{P}$ keeping track of the minimal spanning tree through the graph, have all elements set to ∞. We assume the weighted adjacency matrix $\hat{A}$ of the desired graph has been programmed to a VMM unit before the algorithm begins. This is a one-time cost that can be amortized over frequent reuse of the array. The algorithm then begins by cycling the state machine through the main loop.

ALGORITHM 3: Pseudocode for Temporal Dijkstra's Algorithm

Input: graph G, source node s
// Variable initializations
$\vec{d} := \mathbf{0}_s$; // distances to unvisited nodes (tropical one-hot labels source)
$\vec{v} := \infty$; // visited nodes (tropical zero vector)
$\hat{P} := \infty$; // parent matrix (tropical zero matrix)
$\hat{A} :=$ adjacency-matrix(G); // adjacency matrix of the graph
while $\left(\bigoplus_j d_j < \infty\right)$ **do**
  $\vec{n} :=$ argmin($\vec{d}$); // choose node to visit
  // Examine neighbors
  $\vec{e} := \hat{A} \otimes \vec{n}$; // VMM examine neighbors of current node
  $\vec{f} := \vec{d} \dashv \vec{e}$; // keep only newly found shortest paths
  // Update records for the next iteration
  $\vec{v} := \vec{v} \oplus \vec{n}$; // record the current node as visited
  $\vec{d}' := \vec{d} \oplus \vec{f}$; // construct new record of shortest paths
  $\vec{d} :\cong \vec{v} \dashv \vec{d}'$; // update global unvisited distance vector
  // Parent vector update process
  $\vec{f}^* :\cong$ binarize($\vec{f}$); // vector indices of found nodes
  $\hat{P} := \vec{f}^* \dashv \hat{P}$; // delete row data of previously recorded parents for found nodes
  $\vec{P}_{\vec{n}} := \vec{f}$; // record in column $\vec{n}$ distances $\vec{f}$ from $\vec{n}$ to the found nodes
end
return $\hat{P}$; // adjacency matrix of the minimal spanning (from s) subgraph of G

In each iteration, we check to see if any unvisited nodes are available for exploration by evaluating the minimum element of $\vec{d}$. The algorithm terminates if this operation returns ∞, which indicates that all nodes have either been visited or are unreachable. Taking the argmin (Section 2.5) of $\vec{d}$ nominally gives us a vector $d_j \otimes \mathbf{0}_j$ where j is the index of a node along a shortest path (of those so far explored) from the source and $\mathbf{0}_j$ is the tropical one-hot labeling index j. This result is a one-hot vector, because d_j is always zero by construction. We store this one-hot to the vector register $\vec{n}$.

The next step is to examine the directed edges to the neighbors of node $\vec{n}$. We use $\vec{n}$ as the input to a temporal VMM operation with $\hat{A}$, which performs a parallel traversal to all neighbors. The result is stored in $\vec{e}$, which may contain shorter paths to the neighboring nodes, via node $\vec{n}$, than had been previously found. Such shorter paths would manifest as elements of $\vec{e}$ that have smaller values than their corresponding elements in $\vec{d}$. Those specific nodes can be extracted by taking an elementwise inhibit of $\vec{e}$ by $\vec{d}$; the resulting updated distance vector is stored as $\vec{d}'$. We also note that $\vec{n}$ has been visited, and should not be visited again, by imposing the zero of $\vec{n}$ onto $\vec{v}$ and saving it in memory.

If the dynamic range of our memory were boundless, then we could perform this operation repeatedly and determine the final distance vector of the algorithm. But because we are

dynamic-range-limited,[12] we have to minimize the accumulation in the distance vector. We do this via projective storage of $\vec{d}'$ into $\vec{d}$. We also inhibit $\vec{d}'$ by $\vec{v}$ before storage to ensure that no nodes we have already visited are candidates for exploration in the next iteration; this also ensures argmin($\vec{d}$) will be a magnitude-free one-hot on the next cycle. This shifts the temporal origin for the entirety of the next iteration into the perspective of argmin($\vec{v} \dashv \vec{d}'$); all temporal values in the new $\vec{d}$ are now expressed *relative* to the stopwatch of an observer at the argmin node.

After exploring neighbors, we update the parent matrix. The newly found nodes in $\vec{f}$ need to have their parents updated. A binarized version $\vec{f}^*$ of $\vec{f}$ is used to inhibit rows of the parent matrix corresponding to the new paths in f, erasing these nodes' now-outdated parent data. This operation is performed row-by-row, requiring N state machine transitions to complete. The new parent is then added to the parent matrix; $\vec{n}$ is used to enable the appropriate column of $\hat{P}$ for writing. Vector $\vec{f}$ is then written to this column.

Throughout this algorithm, we require dynamical indexing of memory addresses based on past results of the temporal computation. Recall that Needlemen-Wunsch algorithm required significantly nontrivial subarray selection operations in Algorithm 2. We claimed in Section 3.4 that these would likely need to be handled digitally. Those index selections can be statically determined at compile time, so they could merely be part of the elaborated bytecode controlling the state machine: There is no need for data to translate back and forth between temporal and digital domains to execute Algorithm 2. In Algorithm 3, index selections of the parent matrix are dynamically determined at runtime and cannot be statically embedded in the digital controller around the state machine. But the one-hot nature of the indexing operations offers a natural interface to the crossbar architecture, so, again no digital intermediary is required to perform address lookup.

5 RESULTS

To evaluate this temporal state machine, we make several assumptions in its design and the simulation framework. We create models for temporal memories and the tropical operations required by its design to understand the scaling of this architecture and the tradeoffs of our design space and make predictions about optimization targets.

To achieve realistic first-order performance estimates for this temporal state machine, we design and simulate each component using commercial **very-large-scale integrated circuit (VLSI)** design tools, Cadence Virtuoso for schematic and Spectre for simulation,[13] with experimentally validated nanodevice models [14, 35]. These devices exhibit voltage and current ranges typical of other memristors fabricated by a variety of groups [11, 74, 78]. Though the voltage needed to read these devices can be low (≈200 mV), the voltages needed to write them can be as high as 2 V to 3 V, which puts a lower limit on the technology node we can use. To secure enough voltage headroom for changing device states, we use the 180 nm Silterra process with a V_{dd} = 1.8 V. Though this may not offer the most energy-efficient results, it does provide an understanding of the general set of tradeoffs involved in building realistic temporal state machines. We provide a description of scaling to lower technology nodes by referring to the scaling laws presented in Reference [62]. A discussion on resistive switching technologies at deep-sub-micron nodes is reported in Section 6.2.

[12]Note that even if our memory were not range-limited, we must still choose a dynamic-range cutoff at which we assign finite time values to ∞; otherwise, a circuit that outputs ∞ as a valid return value could never halt. In practice, though, memory is the limiting factor. However, the finite delay representing ∞ is chosen, and it limits the clock frequency of the state machine.

[13]Certain commercial processes and software are identified in this article to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the processes and software identified are necessarily the best available for the purpose.

In this work, the temporal memory is memristive, as is discussed in Reference [46]. The core cell is composed of a 1T1R structure with supporting circuits that allow temporal read and write operations. The temporal read operation is performed by down-shifting the input voltage level from 1.8 V to 600 mV before applying it to the 1T1R array, so the device state is unaffected. This causes the output voltage to have a maximum value of 600 mV, which needs to be up-shifted to 1.8 V for compatibility with other functional blocks, all of which work at V_{dd}. The write path of the memory includes circuits for two different write modes, the conventional and normalized forms described previously. Both these operations require similar circuits with an input first-arrival detector charging the source line and level shifting circuits to the appropriate write voltages, producing the quasilinear state write described in Reference [46].

We have computed read and write energy costs for various $N \times N$ array sizes ranging from $N = 4$ to $N = 32$. The energy scales superlinearly with array size due to growth in support circuitry size that scales with N, the input driver needs to be scaled up for larger array sizes; for the writing, larger array sizes require first-arrival circuitry with more inputs. The read cost is approximately 2 pJ per line, while the write cost is around 10 pJ per line. This 5× factor between read and write energies drives tradeoff considerations in designs.

The most computationally intensive pure race logic function is the tropical VMM, which implements a single-step all-to-all graph traversal. Such an operation naturally scales as N^2. On average, this system ends up costing ≈700 fJ per cell, so a 32×32 grid consumes ≈700 pJ of energy. The large energy cost of this operation arises from the conservative design strategy we employed. To make sure that the OR pulldown network functions properly, we have to ensure that the time constants of the pulldown dynamics are not determined by the CMOS—that is, we have to ensure that it switches quicker than the resolution of our temporal code. The low read voltage causes the pulldown transistor to discharge too slowly, causing multiple nodes pulling down the same source line and leading to functional incorrectness of tropical addition. To overcome this issue, we add level shifters to each cell to boost the input voltage from V_{read} to V_{dd}. These provide the necessary overdrive for correct operation.

Other pure race logic functions such as $\oplus$ = min, $\oplus'$ = max, and $\dashv$ = inhibit, compound functions such as argmin and binarize, and control and multiplexing circuits are implemented with conventional CMOS gates and have a minimal energy cost for this process node. For example, for 32-channel elementwise min, max, and inhibit operations, the energy cost is approximately 1 pJ. This is negligible compared to temporal read, write, and VMM operations. The argmin operation has the largest energy cost among the combinatorial gates, since the first arriving input has to turn of all of the other channels and must therefore drive circuits with a larger output capacitance.

6 COMPARISONS AND TECHNICAL CONSIDERATIONS

6.1 Comparison with Previous Work

Graph processing is a well-studied problem in computing, and a variety of solutions have been proposed for it at various scales [27]. Processing of real world graphs—which can contain hundreds of thousands of nodes and millions of edges—combines both software and hardware frameworks, employing everything from **central processing units (CPUs)**, **field programmable gate arrays (FPGAs)** [79, 82], and **graphics processing units (GPUs)** [23, 72] to **application specific integrated circuits (ASICs)** [29, 75] and **processing-in-memory (PIM)** solutions [12, 61, 81]. Graph operations are known to have a high communication-to-computation ratio, as the cost of memory movement sometimes accounts for upwards of 90% of total energy expenditures. The simple temporal architecture presented in Section 4 is not developed adequately to sensibly compare it to such highly developed systems optimized for much larger graphs. Another impediment to

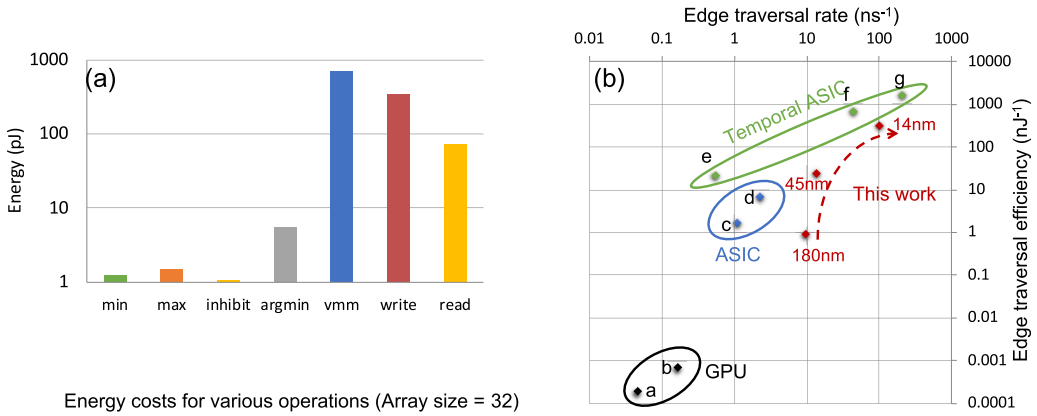

Fig. 7. (a) The energy costs of vector operations for a size 32 array, on a log scale. Basic operations such as min, max, inhibit, and argmin consist only of simple Boolean primitives; consequently, their have low, picojoule energy costs. Their energy scales linearly with the array size. The VMM energy scales quadratically with problem size, involving 1,024 memristive delay elements for the size 32 array. The energy costs of the read and write operations also scale linearly with problem size. (b) The energy cost and inverse latency of a 32×32 temporal kernel and state-of-the-art GPU, ASIC, and temporal ASIC designs. GPU designs (a, b) correspond to MapGraph [23] and Gunrock [72], respectively. ASIC designs (c, d) correspond to Graphicionado [29] and GraphDyn [75]. For both of these ASIC-based approaches, we neglect data movement cost from memory—typically about 90% of the energy expenditure—to allow fair comparisons between kernel operations. Temporal ASIC designs (e, f, g) correspond to wavefront expansion techniques [22], race logic DNA alignment arrays [44], and a time-domain dynamic time warping ASIC [15]. While such approaches show state-of-the-art performance, they are limited to specific graph topologies. Finally, the red points show simulation results from our 180 nm process as well scaling to more advanced nodes following the procedure described in Reference [62].

fair comparison, echoed by the authors of Reference [27], is that much of the extant literature reports relative improvements against other implementations without providing absolute numbers for comparison. This makes comparison with PIM implementations especially difficult. The purpose of this work is to demonstrate the viability of temporal computing as an general approach using a well-studied example, not to compete with the best graph processing engines.

Therefore, we take the following approach: We do not compare against performant CPU and FPGA approaches that leverage 3D-stacked **high-bandwidth memory (HBM)** or **hybrid memory cube (HMC)**, since these approaches rely on the memory management system for their performance advantages. GPU and ASIC approaches with domain-specific kernel implementations amortize the costs of these memory accesses much more effectively and are more popular. For example, MapGraph [23] and Gunrock [72] (points (a, b) in Figure 7(b)) are examples of GPU-based graph analytics packages commonly used as a baseline when reporting performance. More recently, domain-specific accelerators have emerged that have custom datapaths, scheduling strategies, scratchpad memory, and other techniques specifically designed to alleviate the irregularities associated with graph analytics. The literature on these approaches effectively reports the memory versus processing costs [29, 75], allowing us to compare just the performance of our kernel with the performance of other state-of-the-art graph kernels (points (c, d) in Figure 7(b)). Under this analysis, one could imagine swapping in temporal state machines for existing subgraph kernels and measuring changes in overall performance metrics. More recently, temporal ASICs have also been proposed that solve specific graph problems with restricted topologies. Though not designed

for arbitrary graph topologies, they result in superior performance to conventional ASICs. Specific works we compare against are References [15, 22, 44] (points (e, f, g) in Figure 7(b)).

The metric widely used to make speed or latency comparisons is the edge traversal rate, commonly reported as **giga-edge traversals per second (GETS)** in the literature. For energy efficiency, we speak of edges traversed per unit energy—**giga-edge traversals per joule (GETJ)** in the literature. Figure 7(e) shows the performance comparison of this work against GPU and ASIC approaches. A single 32×32 kernel in a 180 nm technology node has an edge traversal rate of 10 ns^{-1} (10 GETS) and the energy efficiency is about 1 nJ^{-1} (1 GETJ), which compares favorably with the state-of-the-art. Using scaling projections from Reference [62], we estimate that a single kernel can theoretically surpass state-of-the-art kernel performance. When scaled up to larger $N \times N$ array sizes, such as $N = 128$ or $N = 256$ (not an uncommon core size for memristor crossbars), we can expect massive performance improvements. Note that the state-of-the-art for graph processing engines when energy is of no concern is on the order of 100s of GETS, which our analysis indicates to be feasible for temporal designs.

Independent but parallel work on graph problems is being undertaken by the neuromorphic computing community. Dijkstra's algorithm has been studied by researchers in neuromorphic computing as a benchmark application for the field [1, 18, 30]. State-of-the-art industrial research spiking neural network platforms [19] use Dijkstra's algorithm to establish performance metrics for their systems. Reference [59] uses single-source shortest path computation to demonstrate their spiking neuromorphic chips and details their energy-per-spike costs: Implementing an operation equivalent to the tropical VMM costs approximately 2.5 nJ in a 65 nm process. By comparison, combining both the memory and VMM primitives, race logic performs the same operation for 1 nJ in a 180 nm process.

6.2 Technical Considerations

6.2.1 Scaling from 180 nm to Newer Technology Nodes. Previous work with race logic has demonstrated that most of the energy expended in race logic architectures is spent in the distribution of timing information, such as in clock trees or analog voltages [44]. To get an energy advantage over those approaches, the present work relies on novel technologies such as memristors to locally generate a programmable delay, which has the advantage that the energy cost is limited by the capacitor. Hence, we are limited by today's memristor technology, which requires large write voltages (1.2 V to 6 V). This requires that we use a relatively old technology node. The development of memristor technology is being driven toward the goal of CMOS compatibility at advanced technology nodes, which require lower read and write voltages [16]. Companies are exploring low write voltage ReRAM and embedding it into 22 nm **fin field-effect transistor (FinFET)** stacks [26, 34].

Maturing technology has great promise for the designs proposed in this work. As CMOS transistors become smaller, the area, energy, and speed all improve. For example, when moving from a 180 nm CMOS to 14 nm FinFET, using a fan-out-of-4 inverter as a benchmark, the area, energy, and latency numbers improve by 100×, 190×, and 19×, respectively [62]. As memristor technologies become compatible with lower voltages, the energy of the read and write operations are expected to decrease. The write energy, determined by the voltage and current needed to alter the memristive state, changes less than the read energy, which follows the inverter characteristics. The scaling performance of race logic systems is easy to estimate, since the spatial nature of the information flow ensures that the architectures in various technology nodes all have similar design and activity factors. We expect the dynamic energy cost to follow the energy trend of the inverter as described in Reference [62]. Though latency and area are determined by other factors such as

memory dynamic range and functional correctness, the overall advantage in energy-delay-product from scaling to a lower node could be as high as three orders of magnitude.

6.2.2 Memristor Device Non-idealities. There are a variety of device non-idealities that affect the design. In the high resistance regime, filamentary memristors suffer from large variations due to the dependence of the tunneling rate on the behavior of only a few ions. Such devices, when measured at low biases, demonstrate random telegraph noise with variability as large as 50% to 100%. In the low-resistance regime, these devices are much more robust, and some groups have demonstrated a programming error of as low as 1.4%, with a closed loop feedback write process [4, 74]. This variability affects the precision and hence the dynamic range that can be encoded with such devices. References [41, 74] demonstrate dynamic ranges of 5 bits to 8 bits. With the memristor model used in this article, we can extract up to 5 bits of precision. Practical implementations have even lower precision. One way to increase precision is to use extra wires to encode higher precision bits as done for Boolean logic. A similar idea has been proposed in References [15, 39]. Future studies may involve analysis of stacking delay-element-based computational units one after another and analyzing the error properties. A preliminary analysis assuming simple statistical properties of device variation indicates that delay-chaining N stages results in an increase of the mean delay by N but of the standard deviation by $\sqrt{N}$. The *relative* variation therefore improves with the number of stages as $1/\sqrt{N}$.

Another impediment to smooth operation in our circuits is the linearity requirement of the memristor write process. A truly linear write would increase the dynamic range of our operations and ensure a clear mapping between the read and write processes. This linearity requirement has been a major topic of research for the neuromorphic VMM community with significant implications for the hardware training of large scale neural networks [10, 63, 74]. Considerable effort has been dedicated to this effort. Various groups show highly linear behavior by operating in the high conductance regime with proper compliance control [41], exploring new materials [13], and using three terminal lithium devices [24]. An alternate approach utilizes highly linear trench-capacitor based storage [42]. Recently, a temporal magnetic memory has been proposed that exhibits linear dynamics [69]. This proposal re-purposes magnetic configurations in racetracks such as domain walls or skyrmions to encode temporal information spatially within the race track.

6.2.3 Future Design Considerations. Scaling to larger graph sizes: Graph processing of large-scale graphs containing trillions of nodes and edges, such as the Internet and social networks, has always been hard to accelerate regardless of hardware choice. Conventional approaches to analyzing these networks rely on graph partitioning, wherein a large graph is partitioned into K subgraphs with a minimal number of edges cut. The K subgraphs can be fit into local processors, with minimal communication between processors. The partitioning problem itself is NP-hard and relies on heuristics for speedup. Various techniques exist for solving such problems: multi-level graph partitioning (involving coarsening, initial partitioning, and fine-tuning) [2], edge or vertex based partitioning [31], breadth-first-search-based partitioning [9], and others. Though graph partitioning is beyond the scope of this work, we intend to extend the circuits described here toward implementation of efficient kernels for accelerating post-partitioned local graph processing.

Chaining of temporal operations: Temporal computation leads to unconventional architectures that come with their own design constraints. The cost of primitive operations (aside from the VMM) in temporal computing is cheap compared to memory access operations. This points to utilizing strategies that amortize the cost of memory accesses over multiple feed-forward operations. Future systems would greatly benefit by performing many such operations in a single phase. In Algorithm 3, for instance, neither $\vec{e}$ nor $\vec{d'}$ need to be stored in memory. A sophisticated

compiler could detect optimally long compositions of pure race logic functions and only use memory where invariance or causality need to be broken. Though such a state machine would need additional control logic with separate clock and dummy lines, the energy savings accrued by this sort of optimization would be significant.

Simple versus complex computational units: As higher-level algorithms become more clearly expressible, an important question would be, what kind of complexity of operations would we want in our designs? A design with simpler fundamental primitives could be more flexible, but might sacrifice performance. An example of that can be seen in the parent matrix update of Algorithm 3. A 2D update array similar to the VMM could amortize the cost of N extra operations, and hence save on N memory reads and writes, in just a single operation. Hence, a more complex operation would have smaller energy and delay, which would be very favorable—at the cost of specialized circuitry. The sensibility of such tradeoffs is an open question that needs to be addressed.

7 CONCLUSION

The utility of temporal computation in solving problems expressible by dynamic programming has been widely noted. Though the first race logic work was proposed as a hardware acceleration for dynamic programming algorithms, it was constrained in its design: a limited topology and a feed-forward memoryless structure. Only the length of the shortest path was reported, with extra circuitry nominally required to report the path itself. Since then, other designs with state-of-the-art performance have been proposed, but they similarly suffer from an *ad hoc* design approach.

In this work, we attempt to make the first steps at generalizability of temporal computing. We provide a problem-agnostic datapath and a mathematical algebra, expanding the logical framework of race logic. This leads to novel circuit designs that are informed by higher-level algorithmic requirements. The properties of abstraction and composability offered by the mathematical framework coupled with native storage from the temporal memory lend themselves to generalization. We design a state machine that can carry out both specialized and general graph algorithms such as the Needleman-Wunsch and Dijkstra's algorithm. The potential for graph accelerators built on temporal computing motivates further exploration of temporal state machines.

ACKNOWLEDGMENTS

Authors thank Brian Hoskins, Mark Anders, Jabez McClelland, Melika Payvand, George Tzimpragos, James Smith, and Tim Sherwood for helpful discussions.

REFERENCES

[1] James B. Aimone, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Helen Xu. 2019. Dynamic programming with spiking neural computing. In *Proceedings of the International Conference on Neuromorphic Systems*. Association for Computing Machinery, New York, NY, 1–9.

[2] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2020. High-quality shared-memory graph partitioning. *IEEE Trans. Parallel Distrib. Syst.* 31, 11 (2020), 2710–2722.

[3] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. 2015. TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 34, 10 (2015), 1537–1557.

[4] Fabien Alibart, Ligang Gao, Brian D. Hoskins, and Dmitri B. Strukov. 2012. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology* 23, 7 (2012), 075201.

[5] Xavier Allamigeon, Pascal Benchimol, Stéphane Gaubert, and Michael Joswig. 2014. Combinatorial simplex algorithms can solve mean payoff games. *SIAM J. Optim.* 24, 4 (2014), 2096–2117.

[6] Xavier Allamigeon, Pascal Benchimol, Stéphane Gaubert, and Michael Joswig. 2015. Tropicalizing the simplex algorithm. *SIAM J. Disc. Math.* 29, 2 (2015), 751–795.

[7] Guo-qiang Bi and Mu-ming Poo. 1998. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *J. Neurosci.* 18, 24 (1998), 10464–10472.

[8] Sander M. Bohte, Joost N. Kok, and Han La Poutre. 2002. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing* 48, 1–4 (2002), 17–37.

[9] Aydin Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patterson. 2017. Distributed-memory breadth-first search on massive graphs. *arXiv:1705.04590* (2017).

[10] Geoffrey W. Burr, Robert M. Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S. Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U. Giacometti et al. 2015. Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses) using phase-change memory as the synaptic weight element. *IEEE Trans. Electron Dev.* 62, 11 (2015), 3498–3507.

[11] Bhaswar Chakrabarti, Miguel Angel Lastras-Montaño, Gina Adam, Mirko Prezioso, Brian Hoskins, M. Payvand, A. Madhavan, A. Ghofrani, L. Theogarajan, K.-T. Cheng, et al. 2017. A multiply-add engine with monolithically integrated 3D memristor crossbar/CMOS hybrid circuit. *Sci. Rep.* 7 (2017), 42429.

[12] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *Proceedings of the ACM/IEEE 47th International Symposium on Computer Architecture (ISCA'20)*. IEEE, 433–445.

[13] Sridhar Chandrasekaran, Firman Mangasa Simanjuntak, R. Saminathan, Debashis Panda, and Tseung-Yuen Tseng. 2019. Improving linearity by introducing Al in HfO2 as a memristor synapse device. *Nanotechnology* 30, 44 (2019), 445205.

[14] Pai-Yu Chen and Shimeng Yu. 2015. Compact modeling of RRAM devices and its applications in 1T1R and 1S1R array design. *IEEE Trans. Electron Dev.* 62, 12 (2015), 4022–4028.

[15] Z. Chen and J. Gu. 2021. High-throughput dynamic time warping accelerator for time-series classification with pipelined mixed-signal time-domain computing. *IEEE J. Solid-state Circ.* 56, 2 (2021), 624–635. DOI: https://doi.org/10.1109/JSSC.2020.3021066

[16] Sumit Choudhary, Mahesh Soni, and Satinder K. Sharma. 2019. Low voltage & controlled switching of MoS2-GO resistive layers based ReRAM for non-volatile memory applications. *Semicond. Sci. Technol.* 34, 8 (2019), 085009.

[17] Simon Davidson, Stephen B. Furber, and Oliver Rhodes. 2020. Spiking associative memory for spatio-temporal patterns. *arXiv:2006.16684* (2020).

[18] Mike Davies. 2019. Benchmarks for progress in neuromorphic computing. *Nat. Mach. Intell.* 1, 9 (2019), 386–388.

[19] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. 2018. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99.

[20] Piotr Dudek. 2006. An asynchronous cellular logic network for trigger-wave image processing on fine-grain massively parallel arrays. *IEEE Trans. Circ. Syst. II: Exp. Briefs* 53, 5 (2006), 354–358.

[21] Tommas J. Ellender, Wiebke Nissen, Laura L. Colgin, Edward O. Mann, and Ole Paulsen. 2010. Priming of hippocampal population bursts by individual perisomatic-targeting interneurons. *J. Neurosci.* 30, 17 (2010), 5979–5991.

[22] Luke R. Everson, Sachin S. Sapatnekar, and Chris H. Kim. 2019. 2.5 A 40× 40 four-neighbor time-based in-memory computing graph ASIC chip featuring wavefront expansion and 2D gradient control. In *Proceedings of the IEEE International Solid-state Circuits Conference (ISSCC'19)*. IEEE, 50–52.

[23] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems*. Association for Computing Machinery, New York, NY, 1–6.

[24] Elliot J. Fuller, Farid El Gabaly, François Léonard, Sapan Agarwal, Steven J. Plimpton, Robin B. Jacobs-Gedrim, Conrad D. James, Matthew J. Marinella, and A. Alec Talin. 2017. Li-ion synaptic transistor for low power analog computing. *Adv. Mater.* 29, 4 (2017), 1604310.

[25] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. 2014. The Spinnaker Project. *Proc. IEEE* 102, 5 (2014), 652–665.

[26] Oleg Golonzka, U. Arslan, P. Bai, M. Bohr, O. Baykan, Y. Chang, A. Chaudhari, A. Chen, N. Das, C. English, et al. 2019. Non-volatile RRAM embedded into 22FFL FinFET technology. In *Proceedings of the Symposium on VLSI Technology*. IEEE, T230–T231.

[27] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *J. Comput. Sci. Technol.* 34, 2 (2019), 339–371.

[28] Rudy Guyonneau, Rufin VanRullen, and Simon J. Thorpe. 2005. Neurons tune to the earliest spikes through STDP. *Neur. Comput.* 17, 4 (2005), 859–879.

[29] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.

[30] Kathleen E. Hamilton, Tiffany M. Mintz, and Catherine D. Schuman. 2019. Spike-based primitives for graph algorithms. *arXiv:1903.10574* (2019).

[31] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed edge partitioning for trillion-edge graphs. *Proc. VLDB Endow.* 12, 13 (2019), 2379–2392.

[32] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[33] Eugene M. Izhikevich. 2006. Polychronization: Computation with spikes. *Neural Comput.* 18, 2 (2006), 245–282.

[34] Pulkit Jain, Umut Arslan, Meenakshi Sekhar, Blake C. Lin, Liqiong Wei, Tanaya Sahu, Juan Alzate-Vinasco, Ajay Vangapaty, Mesut Meterelliyoz, Nathan Strutt, et al. 2019. 13.2 A 3.6 Mb 10.1 Mb/mm 2 embedded non-volatile ReRAM macro in 22nm FinFET technology with adaptive forming/set/reset schemes yielding down to 0.5 V with sensing time of 5ns at 0.7 V. In *Proceedings of the IEEE International Solid-state Circuits Conference (ISSCC'19)*. IEEE, 212–214.

[35] Zizhen Jiang, Shimeng Yu, Yi Wu, Jesse H. Engel, Ximeng Guan, and H.-S. Philip Wong. 2014. Verilog—A compact model for oxide-based resistive random access memory (RRAM). In *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD'14)*. IEEE, 41–44.

[36] Bill Kay, Prasanna Date, and Catherine Schuman. 2020. Neuromorphic graph algorithms: Extracting longest shortest paths and minimum spanning trees. In *Proceedings of the Neuro-inspired Computational Elements Workshop*. Association for Computing Machinery, New York, NY, 1–6.

[37] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'16)*. IEEE, 1–9.

[38] Dion Khodagholy, Jennifer N. Gelinas, Thomas Thesen, Werner Doyle, Orrin Devinsky, George G. Malliaras, and György Buzsáki. 2015. NeuroGrid: Recording action potentials from the surface of the brain. *Nat. Neurosci.* 18, 2 (2015), 310–315.

[39] KwangSeok Kim, Wonsik Yu, and SeongHwan Cho. 2014. A 9 bit, 1.12 ps resolution 2.5 b/stage pipelined time-to-digital converter in 65 nm CMOS using time-register. *IEEE J. Solid-state Circ.* 49, 4 (2014), 1007–1016.

[40] Xavier Lagorce and Ryad Benosman. 2015. STICK: Spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony. *Neural Comput.* 27, 11 (2015), 2261–2317.

[41] Can Li, Miao Hu, Yunning Li, Hao Jiang, Ning Ge, Eric Montgomery, Jiaming Zhang, Wenhao Song, Noraica Dávila, Catherine E. Graves, et al. 2018. Analogue signal and image processing with large memristor crossbars. *Nature Electron.* 1, 1 (2018), 52.

[42] Y. Li, S. Kim, X. Sun, P. Solomon, T. Gokmen, H. Tsai, S. Koswatta, Z. Ren, R. Mo, C. C. Yeh, et al. 2018. Capacitor-based cross-point array for analog neural network with record symmetry and linearity. In *Proceedings of the IEEE Symposium on VLSI Technology*. IEEE, 25–26.

[43] D. P. Lopresti and R. J. Lipton.1985. A systolic array for rapid string comparison. In *Proceedings of the Chapel Hill Conference on VLSI, 1985*. Computer Science Press, Chapel Hill, NC, 363–376.

[44] Advait Madhavan, Timothy Sherwood, and D. Strukov. 2017. A 4-mm 2 180-nm-CMOS 15-Giga-cell-updates-per-second DNA sequence alignment engine based on asynchronous race conditions. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC'17)*. IEEE, 1–4.

[45] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2014. Race logic: A hardware acceleration for dynamic programming algorithms. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. IEEE, 517–528.

[46] A. Madhavan and M. D. Stiles. 2020. Storing and retrieving wavefronts with resistive temporal memory. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'20)*. IEEE, 1–5. DOI: https://doi.org/10.1109/ISCAS45731.2020.9180662.

[47] Advait Madhavan, Georgios Tzimpragos, Mark Stiles, and Timothy Sherwood. 2019. A truth-matrix view into unary computing. In *Proceedings of the 1st Unary Computing Workshop (ISCA'19)*.

[48] Daisuke Miyashita, Shouhei Kousai, Tomoya Suzuki, and Jun Deguchi. 2017. A neuromorphic chip optimized for deep learning and CMOS technology with time-domain analog and digital mixed-signal processing. *IEEE J. Solid-state Circ.* 52, 10 (2017), 2679–2689.

[49] Mehryar Mohri. 2002. Semiring frameworks and algorithms for shortest-distance problems. *J. Autom., Lang. Combinat.* 7, 3 (2002), 321–350.

[50] Harideep Nair, John Paul Shen, and James E. Smith. 2020. Direct CMOS Implementation of Neuromorphic Temporal Neural Networks for Sensory Processing. arxiv:cs.AR/2009.00457 (2020).

[51] M. Hassan Najafi, David J. Lilja, Marc D. Riedel, and Kia Bazargan. 2018. Low-cost sorting network circuits using unary processing. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 26, 8 (2018), 1471–1480.

[52] Taehwan Oh, Hariprasath Venkatram, and Un-Ku Moon. 2013. A time-based pipelined ADC using both voltage and time domain information. *IEEE J. Solid-state Circ.* 49, 4 (2013), 961–971.

[53] Garrick Orchard, Cedric Meyer, Ralph Etienne-Cummings, Christoph Posch, Nitish Thakor, and Ryad Benosman. 2015. HFirst: A temporal approach to object recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 10 (2015), 2028–2040.

[54] Marc Osswald, Sio-Hoi Ieng, Ryad Benosman, and Giacomo Indiveri. 2017. A spiking neural network model of 3D perception for event-based neuromorphic stereo vision systems. *Sci. Rep.* 7 (2017), 40703.

[55] Filip Ponulak and Andrzej Kasiński. 2010. Supervised learning in spiking neural networks with ReSuMe: Sequence learning, classification, and spike shifting. *Neural Comput.* 22, 2 (2010), 467–510.

[56] Filip Jan Ponulak and John J. Hopfield. 2013. Rapid, parallel path planning by propagating wavefronts of spiking neural activity. *Front. Comput. Neurosci.* 7 (2013), 98.

[57] Shubham Sahay, Mohammad Bavandpour, Mohammad Reza Mahmoodi, and Dmitri Strukov. 2020. A 2T-1R cell array with high dynamic range for mismatch-robust and efficient neurocomputing. In *Proceedings of the IEEE International Memory Workshop (IMW'20)*. IEEE, 1–4.

[58] Aseem Sayal, S. S. Teja Nibhanupudi, Shirin Fathima, and Jaydeep P. Kulkarni. 2019. A 12.08-TOPS/W all-digital time-domain CNN engine using bi-directional memory delay lines for energy efficient edge computing. *IEEE J. Solid-state Circ.* 55, 1 (2019), 60–75.

[59] Catherine D. Schuman, Kathleen Hamilton, Tiffany Mintz, Md Musabbir Adnan, Bon Woong Ku, Sung-Kyu Lim, and Garrett S. Rose. 2019. Shortest path and neighborhood subgraph extraction on a spiking memristive neuromorphic implementation. In *Proceedings of the 7th Neuro-inspired Computational Elements Workshop*. Association for Computing Machinery, New York, NY, 1–6.

[60] James E. Smith. 2018. Space-time algebra: A model for neocortical computation. In *Proceedings of the 45th International Symposium on Computer Architecture*. IEEE Press, Los Angeles, CA, 289–300.

[61] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 531–543.

[62] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[63] Xiaoyu Sun and Shimeng Yu. 2019. Impact of non-ideal characteristics of resistive synaptic devices on implementing convolutional neural networks. *IEEE J. Emerg. Select. Topics Circ. Syst.* 9, 3 (2019), 570–579.

[64] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. 2019. Deep learning in spiking neural networks. *Neural Netw.* 111 (2019), 47–63.

[65] Simon J. Thorpe. 1990. Spike arrival times: A highly efficient coding scheme for neural networks. In *Parallel Processing in Neural Systems and Computers*, R. Eckmiller, G. Hartmann, and G. Hauske (Eds.). North Holland Elsevier, Amsterdam, 91–94.

[66] Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. 2019. Boosted race trees for low energy classification. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 215–228.

[67] Georgios Tzimpragos, Nestan Tsiskaridze, Kylie Huch, Advait Madhavan, and Timothy Sherwood. 2019. From arbitrary functions to space-time implementations. In *Proceedings of the 1st Unary Computing Workshop (ISCA'19)*.

[68] Georgios Tzimpragos, Dilip Vasudevan, Nestan Tsiskaridze, George Michelogiannakis, Advait Madhavan, Jennifer Volk, John Shalf, and Timothy Sherwood. 2020. A computational temporal logic for superconducting accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 435–448.

[69] Hamed Vakili, Mohammad Nazmus Sakib, Samiran Ganguly, Mircea Stan, Matthew W. Daniels, Advait Madhavan, Mark D. Stiles, and Avik W. Ghosh. 2020. Temporal memory with magnetic racetracks. *J. Expl. Sol.-state Comp. Dev. and Circ.* 6, 2 (2020), 107–115.

[70] Rufin VanRullen, Rudy Guyonneau, and Simon J. Thorpe. 2005. Spike times make sense. *Trends Neurosci.* 28, 1 (2005), 1–4.

[71] Stephen J. Verzi, Fredrick Rothganger, Ojas D. Parekh, Tu-Thach Quach, Nadine E. Miner, Craig M. Vineyard, Conrad D. James, and James B. Aimone. 2018. Computing with spikes: The advantage of fine-grained timing. *Neural Comput.* 30, 10 (2018), 2660–2690.

[72] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, 1–12.

[73] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. 2018. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Front. Neurosci.* 12 (2018), 331.

[74] Qiangfei Xia and J. Joshua Yang. 2019. Memristive crossbar arrays for brain-inspired computing. *Nat. Mater.* 18, 4 (2019), 309–323.

[75] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng et al. 2019. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, 615–628.

[76] Heemin Y. Yang and Rahul Sarpeshkar. 2005. A time-based energy-efficient analog-to-digital converter. *IEEE J. Solid-tate Circ.* 40, 8 (2005), 1590–1601.

[77] Ruriko Yoshida, Leon Zhang, and Xu Zhang. 2019. Tropical principal component analysis and its application to phylogenetics. *Bull. Math. Biol.* 81, 2 (2019), 568–597.

[78] Shimeng Yu. 2018. Neuro-inspired computing with emerging nonvolatile memorys. *Proc. IEEE* 106, 2 (2018), 260–285.

[79] Jialiang Zhang and Jing Li. 2018. Degree-aware hybrid graph traversal on FPGA-HMC platform. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. Association for Computing Machinery, New York, NY, 229–238.

[80] Liwen Zhang, Gregory Naitzat, and Lek-Heng Lim. 2018. Tropical geometry of deep neural networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 5824–5832. Retrieved from http://proceedings.mlr.press/v80/zhang18i.html.

[81] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: Graph processing in a ReRAM-based computational memory. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. Association for Computing Machinery, New York, NY, 591–596.

[82] Shijie Zhou and Viktor K. Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'17)*. IEEE, 137–144.

Received September 2020; revised February 2021; accepted February 2021