SIMD||DNA: Single Instruction, Multiple Data Computation with DNA Strand Displacement Cascades

Boya Wang, Cameron Chalk, and David Soloveichik

The University of Texas at Austin

Abstract. Typical DNA storage schemes do not allow in-memory computation, and instead transformation of the stored data requires DNA sequencing, electronic computation of the transformation, followed by synthesizing new DNA. In contrast we propose a model of in-memory computation that avoids the time consuming and expensive sequencing and synthesis steps, with computation carried out by DNA strand displacement. We demonstrate the flexibility of our approach by developing schemes for massively parallel binary counting and elementary cellular automaton Rule 110 computation.

Keywords: DNA storage · DNA computing · parallel computing · strand displacement

1 Introduction

Studies have espoused DNA as an incredibly dense (up to 455 exabytes per gram) and stable (readable over millenia) digital storage medium [5]. Experiments storing text, images, and movies of hundreds of megabytes have demonstrated the potential scalability of the approach [11]. Importantly, DNA's essential biological role ensures that the technology for manipulating DNA will never succumb to obsolescence.

Typical DNA storage schemes have high information density but do not permit "in-memory" computation: modifying data involves sequencing DNA, classically computing the desired transformation, and synthesizing new DNA. In contrast, strand displacement systems store information in the pattern of reconfiguration of exposed single-stranded regions. This pattern can be directly manipulated through toehold exchange and other molecular primitives as a form of information processing [25]. However, strand displacement is incompatible with traditional DNA storage schemes.

Here we combine DNA storage with massively parallel computation on the data stored using strand displacement. In our proposed scheme, which we call SIMD||DNA (Single Instruction Multiple Data DNA), a multi-stranded DNA complex acts as a single register storing a (binary) string. Although all the complexes share the same sequence, different information is encoded in each complex in the pattern of nicks and exposed single-stranded regions. There are as many independent registers as the number of molecules of the multi-stranded complexes, each capable of storing and manipulating a different string. This allows information in different registers to be modified at the same time, utilizing the parallelism granted by molecular computation.

Our method of storing information in DNA is motivated by recent developments in DNA storage employing topological modifications of DNA to encode data. DNA storage based on programmable nicking on native DNA (forming strand breaks at desired locations) permits high throughput methods of writing information into registers [20]. To enable subsequent read-out, recently developed methods [9] could potentially read information encoded in nicks and single-stranded gaps in double stranded DNA in a high throughput manner. Reading out specific bits of registers could also be achieved with fluorescence based methods. Note that compared with storing data in the DNA sequence itself, encoding data in nicks sacrifices data density but reduces the cost of large-scale synthesis of DNA [20]. Here we show that it also enables greater flexibility of in-memory computation.

To do parallel in-memory computation on our DNA registers, we employ single instruction, multiple data (SIMD)[1] programs. An overview of a program's implementation is given in Figure 1. Each instruction of a program corresponds to the addition of a set of DNA strands to the solution. The added strands undergo toehold-mediated strand displacement with strands bound to the register, changing the data. The long "bottom" strands of these registers are attached to magnetic beads, allowing sequential elution operations. After the strands displaced from the registers are eluted, subsequent instructions can be performed. Note that the same instruction is applied to all registers in solution in parallel (since they share sequence space), but the effect of that instruction can be different depending on the pattern of nicks and exposed regions of the given register.

We show that our DNA data processing scheme is capable of parallel, inmemory computation, eliminating the need for sequencing and synthesizing new DNA on each data update. Note that instruction strands are synthesized independently of the data stored in the registers, so that executing an instruction does not require reading the data. We should also note the doubly-parallel nature of SIMD||DNA programs: instructions act on all registers in parallel, and instruction strands can act on multiple sites within a register in parallel.

Our programs require a small number of unique domains (subsequences of nucleotides which act as functional units), independent of the register length.

[1] Single instruction, multiple data (SIMD) is one of the four classifications in Flynn's taxonomy [7]. The taxonomy captures computer architecture designs and their parallelism. The four classifications are the four choices of combining single instruction (SI) or multiple instruction (MI) with single data (SD) or multiple data (MD). SI versus MI captures the number of processors/instructions modifying the data at a given time. SD versus MD captures the number of data registers being modified at a given time, each of which can store different information. Our scheme falls under SIMD, since many registers, each with different data, are affected by the same instruction.

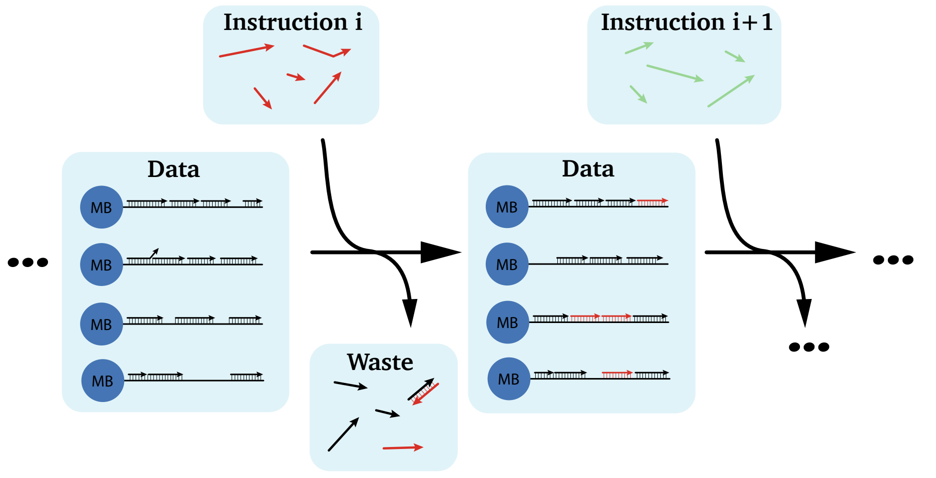

Fig. 1: Each DNA register is a multi-stranded complex. Different information is encoded in the pattern of nicks and exposed single-stranded regions in the register. Registers are attached to magnetic beads (MB). To perform each instruction, first a set of instruction strands is added to the solution and reacts with all the registers in parallel. Then waste species (unreacted instruction strands and displaced reaction products) are washed away by elution.

A common assumption for correctness in strand displacement systems is that domains are *orthogonal*, meaning two domains which are not fully complementary do not bind. In experiments, enforcing this assumption requires specialized sequence design. Further, for any fixed domain length, the space of orthogonal domains is limited, restricting the scalability of the system. SIMD||DNA encodes information in the pattern of nicks and exposed domains. This allows our programs to require only a constant set of orthogonal domains to be used (five for one program and six for the other), simplifying the sequence design problem for experimental implementation. In addition, the instruction strands for one instruction can share sequences, resulting in a reduced cost of strand synthesis.

In this paper, we show two SIMD||DNA programs. One of the programs implements binary counting. Starting from arbitrary initial counts stored in different registers, each computation step increments all the registers in parallel. Binary counting allows one SIMD||DNA program to move data through a number of states exponential in the size of the register. We consider this a requirement of any useful data storage/computation suite: if instead not all configurations of the register were reachable from some initial configuration via some program, then the useful density of the storage would be reduced.

In addition to binary counting, we also give a program which simulates elementary cellular automaton (CA) Rule 110.[2] Critically, Rule 110 has been

[2] In [24] an enumeration of all possible rules for elementary CA is given. Rule 110 refers to that enumeration.

shown to be Turing universal [6], so this simulation shows that SIMD||DNA's in-memory computation model is as powerful as any other space-bounded computing technique. In other words, our space-bounded simulation of Rule 110 immediately gives that any computable function—if the required space is known beforehand—can be computed by a SIMD||DNA program.

We note the contrast to typical strand displacement schemes that perform a single computation in solution. For example, although a logic circuit [18, 13] computation might involve hundreds of billions of separate molecules, the redundancy does not help computationally. Such schemes seem to not use the massively parallel power of chemistry [1]. Previous ideas for performing parallel computation with strand displacement cascades relied on a complex scheme involving 4-way branch migration on DNA origami [14] or information processing in different spatial locations [17]. Turing universal computation with strand displacement could not handle multiple independent computations running in parallel [12], leaving the extension to parallel computation as the major open question.

Note that SIMD||DNA is non-autonomous since each instruction requires manual strand addition and elution. In this regard it is similar to early studies of parallel DNA computing machines. Dating back to the 1990s, Adleman experimentally demonstrated solving instances of NP-complete problems using DNA [1], which encouraged other DNA computing models. Many models rely on enzymes to introduce covalent modification on DNA [2, 3, 8, 15], which increases experimental complexity. Other enzyme-free models such as the sticker model [16] encode information in the pattern of exposed domains, similar to our scheme. However, the sticker model requires a number of orthogonal domain types that scales with the amount of data. In addition, these domains require well-tuned binding affinities to allow a melting procedure which selectively dissociates some strands but not others. In contrast, our programs only require a constant number of unique domains for any register length. Instead of computation through controlled hybridization and melting, strand displacement is a more versatile mechanism to achieve modification of information, potentially making parallel molecular computation more feasible.[3]

2 SIMD||DNA

Here we propose the general scheme. First we will explain the notations we use in this paper. We use the domain level abstraction for DNA strands. Consecutive nucleotides that act as a functional unit are called a domain. Complementary domains are represented by a star ($*$). The length of the domains is chosen so that: (1) each domain can initiate strand displacement (can act as a toehold), (2)

[3] In a sense, we realize an extension of the sticker model envisioned by [4]: "Recent research suggests that DNA 'strand invasion' might provide a means for the specific removal of stickers from library strands. This could give rise to library strands that act as very powerful read-write memories. Further investigation of this possibility seems worthwhile."

strands bound by a single domain readily dissociate, and (3) strands bound by two or more domains cannot dissociate.[4] We call an exposed (unbound) domain a *toehold*.

2.1 Encoding data

Data is stored in multi-stranded complexes (Figure 1), each called a register. A register contains one long strand, called the *bottom strand* and multiple short strands, called *top strands*, bound to the bottom strand. Each bottom strand is partitioned into sets of consecutive domains called *cells*. Each cell contains the same number of domains. Depending on the configuration of the top strands bound (e.g., their lengths, or the presence or absence of toeholds), cells encode information. In this work we use a binary encoding, with each cell representing one bit.

See Section 4.1 for a discussion of potential experimental methods of preparing the initial registers.

2.2 Instructions

An *instruction* is a set of strands. To apply an instruction to the data, these strands are added to the solution at high concentration. Adding these strands can lead to three different types of reactions on the registers. Figure 2a explains the figure notation used to describe instructions throughout the paper, and Figure 2b gives examples of the three types of reactions. They are:

Attachment: This reaction preserves all the strands originally bound to the register and attaches new strands. An instruction strand can attach to registers if it binds strongly enough (by two or more domains). Note that the attachment of an instruction strand can lead to a partial displacement of a pre-existing strand on the register.

Displacement: This reaction introduces new strands to the register and detaches some pre-existing strands. Upon binding to a toehold on the register, the instruction strand displaces pre-existing strands through 3-way branch migration.[5] Toehold exchange reactions are favored towards displacement by the instruction strand since they are added at high concentration. Two instruction strands can also cooperatively displace strands on the register.

Detachment: This reaction detaches pre-existing strands without introducing new strands to the registers. An instruction strand that is complementary to a pre-existing strand with an open overhang can use the overhang as a toehold and pull the strand off the register. Throughout this paper, a dashed instruction

[4] Given these properties, in practice one could choose the domain length to be from 5 to 7 nucleotides at room temperature.

[5] Although other more complicated strand displacement mechanisms (e.g. 4-way, remote toehold, associative toehold strand displacement) could provide extra power in this architecture, they usually sacrifice the speed and increase the design complexity, so we do not include them in this work.

strand indicates the domains in the instruction strand are complementary to other vertically aligned domains.

When an instruction strand displaces a top strand, we assume the waste top strand does not interact further within the system (the instruction strands are present in high concentration while the waste is in low concentration). After the reactions complete, the waste is removed via elution. We assume washing removes all species without a magnetic bead. Lastly, we assume there is no *leak*—displacement of a strand without prior binding of a toehold. We discuss the possibility and extent of errors caused by relaxing these assumptions in Section 4.2.

In general, two reactions can be applicable but mutually exclusive. Then two (or more) resulting register states may be possible after adding the instruction strands. The instructions used in this paper do not have this issue. This point is related to *deterministic* versus *nondeterministic* algorithms, and is discussed further in Section 4.5.

2.3 Programs

We consider sequences of instructions, called *programs*. We design programs for functions $f : \{0,1\}^n \to \{0,1\}^n$ so that, given a register encoding any $s = \{0,1\}^n$, after applying all instructions in the program sequentially as in Figure 1, the resulting register encodes $f(s)$.

3 Programs for binary counting and Rule 110

Here we give our two programs: binary counting and simulation of elementary cellular automaton Rule 110. We first present the Rule 110 simulation, as the program is simpler to explain than binary counting.

3.1 Cellular Automaton Rule 110

An elementary one-dimensional cellular automaton consists of an infinite set of cells $\{\ldots, c_{-1}, c_0, c_1, \ldots\}$. Each cell is in one of two states, 0 or 1. Each cell changes state in each timestep depending on its left and right neighbor's states. Rule 110 is defined as follows: the state of a cell at time $t+1$, denoted $c_i(t+1)$, is $f(c_{i-1}(t), c_i(t), c_{i+1}(t))$, where f is the following:

$$\begin{aligned} f(0,0,0) &= 0 & \qquad f(1,0,0) &= 0 \\ f(0,0,1) &= 1 & \qquad f(1,0,1) &= 1 \\ f(0,1,0) &= 1 & \qquad f(1,1,0) &= 1 \\ f(0,1,1) &= 1 & \qquad f(1,1,1) &= 0 \end{aligned}$$

Note that a simple two-rule characterization of f is as follows: 0 updates to 1 if and only if the state to its right is a 1, and 1 updates to 0 if and only if both

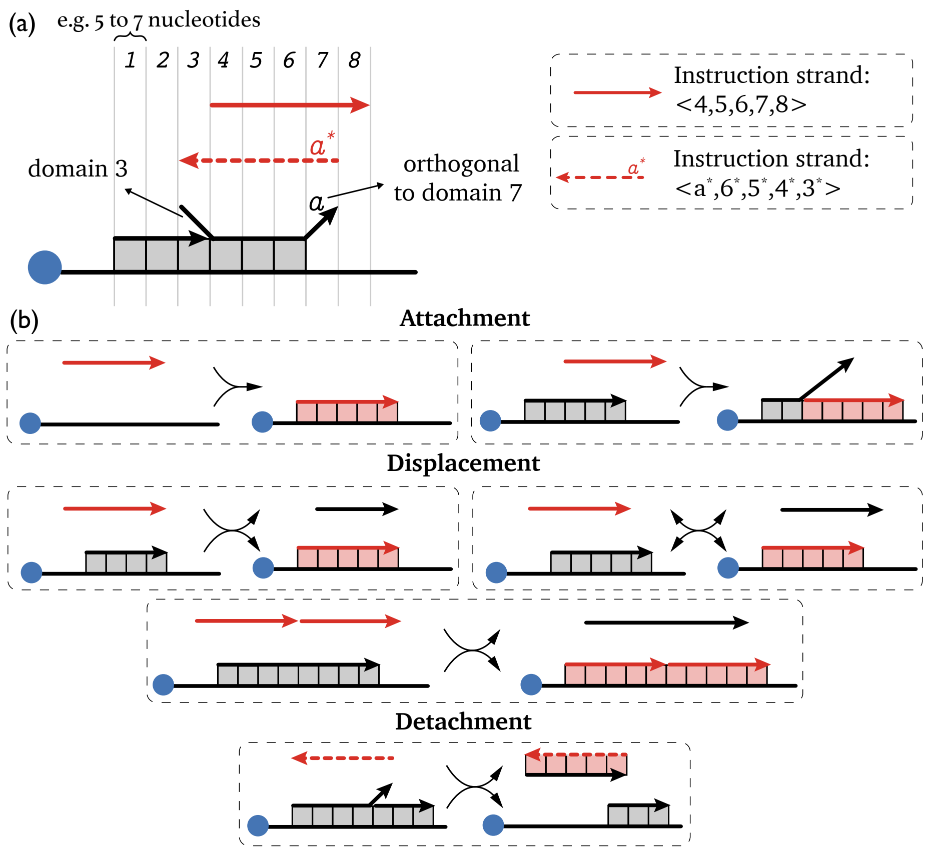

Fig. 2: (a) The notation used to describe instructions. Domains are represented by square boxes. We indicate complementarity of instruction strands to register domains by vertical alignment. If a domain label is given explicitly, such as a and a^* in this figure, the domain is orthogonal to the other vertically aligned domains. A strand can be described by listing the constituent domains in a bracket <> from 5'-end to 3'-end. Strands with solid lines are complementary to the corresponding domains in the bottom strand. Strands with dashed lines are complementary to the corresponding domains in the top strand. The blue dot represents the magnetic bead. (b) The three instruction reactions. Attachment: instruction strands attach to the register without releasing any other strands. Displacement: instruction strands attach to the register and displace pre-existing strands on the register. Toehold-mediated strand displacement (left), toehold-exchange strand displacement (right), and cooperative strand displacement (bottom) mechanisms are allowed. Detachment: instruction strands fully complementary to previously bound top strands pull strands off the register.

neighbors are 1. This characterization is useful for proving correctness of the program.

The instructions implementing one timestep evolution are shown in Figure 3. Each state-0 cell is fully covered by two strands, one of length three and one of length two. Each state-1 cell is partially covered by a length-five top strand and has an open toehold at the leftmost domain. The program consists of six instructions. The program first marks the string "01" (Instruction 1)—here, the 0 will change to 1 later. Then it erases the internal 1's in any string of at least three consecutive 1's (Instructions 2 and 3). These are the 1's with two neighboring 1's, which should be updated to 0, so the program fills in the empty cells with 0 (Instruction 4). Finally it removes the markers from Instruction 1 and changes previously marked 0's to 1's (Instructions 5 and 6).

We claim that this program enforces the two-rule characterization of Rule 110. We first argue that 1 updates to 0 if and only if both neighbors are 1. Then we argue that 0 updates to 1 if and only if the state to its right is a 1. Let i_k denote the kth domain on cell i (from left to right). All cells can share the same sequences, but we assume that each domain within a cell is orthogonal.

Claim. A cell i initially in state 1 updates to a 0 if cells $i+1$ and $i-1$ are initially 1.

Proof. During Instruction 1, the instruction strands cannot displace the strands in state-1 cells. In Instruction 2, the strand on cell i is displaced cooperatively only if the toeholds on both the left and the right side of the strand are open. By assumption, cell $i+1$ is a 1, so the toehold immediately to the right of cell i, $(i+1)_1$, is free. Since cell $i-1$ is in state 1, domain i_1 is not covered after Instruction 1 (i_1 would be covered if cell $i-1$ were 0). Thus the strand on cell i can be displaced by the instruction 2 strands. In Instruction 3, the instruction 2 strands in cell i are detached, so every domain in cell i is free. Then in Instruction 4 we attach the strands corresponding to a state 0, updating cell i to 0. Instructions 5 and 6 do not introduce any instruction reaction on cell i, so cell i remains in state 0. □

Claim. A cell i initially in state 1 stays in state 1 if either cell $i+1$ or $i-1$ is initially 0.

Proof. During Instruction 1, the instruction strands cannot displace the strands in state-1 cells. In Instruction 2, the strand on state-1 cells is displaced cooperatively only if the toeholds on both the left and the right side of the strand are open. By assumption that the left or right cell is a 0, the toeholds required for this, i_1 or $(i+1)_1$, will be covered: First consider that cell $i-1$ is a 0. Then in Instruction 1, the instruction strand displaces one strand at cell $i-1$ and covers the toehold i_1. On the other hand, if cell $i+1$ is 0, then domain $(i+1)_1$ is covered since strands for the 0 state cover all the domains in that cell. So if either neighbor state is 0, Instruction 2 does not displace the strand on cell i. Then note that Instructions 3 and 4 do not introduce any instruction reaction at cell i. The instruction 5 strands detach the instruction 1 strands if cell $i-1$

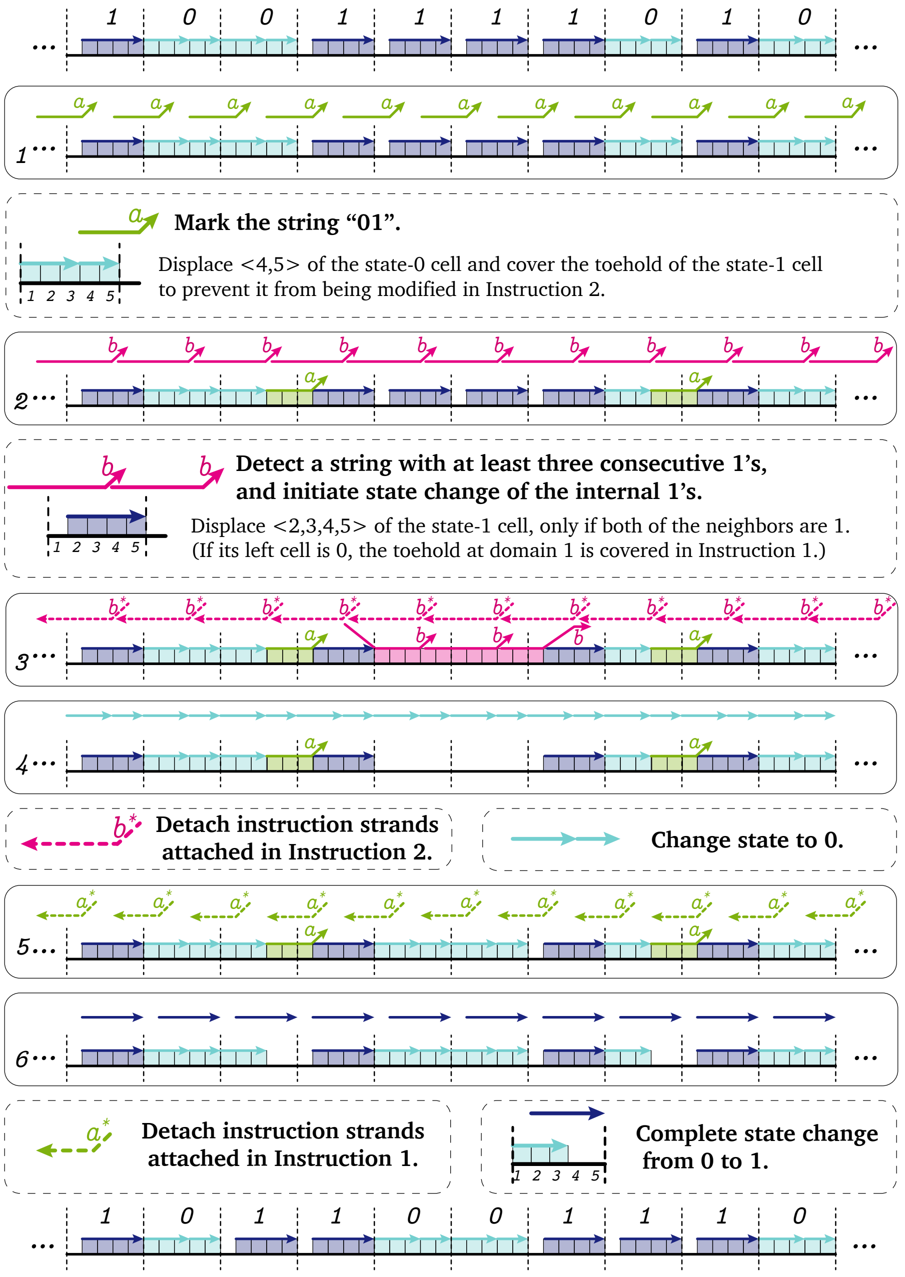
1 0 0 1 1 1 1 0 1 0
1
a
Mark the string "01".
Displace <4,5> of the state-0 cell and cover the toehold of the state-1 cell to prevent it from being modified in Instruction 2.
1 2 3 4 5
2
b
Detect a string with at least three consecutive 1's, and initiate state change of the internal 1's.
Displace <2,3,4,5> of the state-1 cell, only if both of the neighbors are 1. (If its left cell is 0, the toehold at domain 1 is covered in Instruction 1.)
3
b*
4
Detach instruction strands attached in Instruction 2.
Change state to 0.
5
a*
6
Detach instruction strands attached in Instruction 1.
Complete state change from 0 to 1.
1 0 1 1 0 0 1 1 1 0

Fig. 3: The program implementing one timestep of Rule 110 shown on an example register. The top register shows the initial state of each cell. After 6 instructions, the register updates to the state shown at the bottom. Strand colors have three information categories: state 1 (dark blue), state 0 (light blue), intermediates (other colors). Solid boxes show the instruction strands and the state of the register before the strands are applied. Dashed boxes explain the logical meaning of the instructions. The overhang domains a and b are orthogonal to their vertically aligned domains.

is 0, freeing the toehold at i_1 and recovering the state-1 cell. Instruction 6 does not change the state-1 cell. □

Claim. A cell i initially in state 0 updates to a 1 if cell $i+1$ is initially a 1.

Proof. Since cell $i+1$ is in state 1, the toehold at domain $(i+1)_1$ is available for the instruction strand in Instruction 1 to bind, and the rightmost strand on cell i is displaced. Then note that Instructions 2 through 4 do not introduce any instruction reaction at cell i. In Instruction 5, the instruction strand from Instruction 1 is detached, freeing domains i_4 and i_5. In Instruction 6 the instruction strand binds at domains i_4 and i_5 and displaces the strand at cell i. So after Instruction 6, cell i is in state 1. □

Claim. A cell i initially in state 0 stays in state 0 if cell $i+1$ is initially a 0.

Proof. Simply note that for any instruction, no instruction reaction on cell i occurs. So cell i stays in state 0. □

These four claims combined verify that the two-rule characterization given at the beginning of this section is satisfied, so the instructions implement one timestep evolution of Rule 110.

Note that the Rule 110 simulation invokes two sources of parallelism. Instruction strands are applied to all registers in parallel, and every cell within a register can update concurrently.

Also note that Rule 110 is defined only for an infinite set of cells or a circular arrangement of finitely many cells. For a finite set of cells arranged linearly, one must define boundary conditions for updating the leftmost and rightmost cells. Boundary conditions can be constant or periodic. For space-bounded computation by Rule 110, it suffices to set periodic boundary conditions based on the periodic initial condition of the CA given in [6]. These periodic boundary states can be implemented by periodic instructions.

3.2 Counting

The counting program computes $f(s) = s + 1$. Binary counting is captured by changing all the 1s to 0 from the least significant bit to more significant bits until the first 0, and changing that 0 to 1. All the bits more significant than the rightmost 0 remain the same. For example, $f(1011) = 1100$, and $f(1000) = 1001$. In the case of overflow, we rewrite the register to all 0s. In other words, on inputs of all 1s, we output all 0s: $f(1111) = f(0000)$.

The full program is in Figure 4. Each state-0 cell is fully covered by two strands, with one covering the first three domains and the other one covering the last two domains. Each state-1 cell is fully covered by two strands, with one covering the first two domains and the other one covering the last three domains. One extra domain is included to the right of the rightmost cell which is used to initiate displacement. The program contains seven instructions. It erases all the 1's in between the rightmost cell and the rightmost state-0 cell at Instructions

1 and 2, and changes those cells to 0 at Instructions 4 and 5. It marks the rightmost state-0 cell at Instruction 3, and change the marked state-0 cell to state 1 at Instructions 6 and 7.

To prove correctness, we first argue that all the 1's from the least significant bit to the rightmost 0 update to 0. Then we argue that rightmost 0 updates to 1. Assume the bit string has length n and the least significant bit is at cell n and the rightmost 0 is at cell m ($m \leqslant n$). As in the Rule 110 simulation proof, we let j_k denote the kth domain on cell j (from left to right). All cells can share the same sequences, but we assume that each domain within a cell is orthogonal. Additionally, the extra domain to the right of the rightmost cell is orthogonal to all other domains.

Claim. All state 1 cells to the right of the rightmost 0 cell change to a 0.

Proof. Instruction 1 initiates a series of sequential reactions from the least significant bit n to the rightmost 0. First the instruction strand with overhang domain a displaces the strand covering domains n_4 and n_5. If the least significant bit is 1 ($m < n$), the domain n_3 becomes unbound after this displacement reaction. Then the domain n_3 serves as an open toehold to initiate another displacement reaction with the instruction strand with overhang domain b. Similar displacement reactions proceed until cell m. By assumption, cell m is a state-0 cell, so the domain m_3 will not be open after the displacement step, thus the displacement cascade stops. Then the strands added in Instruction 2 detach the strands from Instruction 1, leaving the cells from the $(m + 1)$th bit to the nth bit free. In Instruction 3, every applied instruction strand from cell $m + 1$ to n attaches to the register. Instruction 4 shifts those strands added in Instruction 3 one domain to the left, which opens toeholds for the cooperative displacement in Instruction 5. After those cells change to state-0 in Instruction 5, the strands added in Instruction 6 and 7 do not change them, so they remain in state 0. □

Claim. The rightmost state 0 cell changes to a 1.

Proof. Instruction 1 initiates a series of sequential reactions from the least significant bit to the rightmost 0 at cell m. The domain m_3 will not be open after the instruction strand displaces the strand covering domains m_4 and m_5 and no more strand displacement cascade can proceed to the left. Then the strands added in Instruction 2 detach the strands from Instruction 1, leaving the domains m_4 and m_5 free. The strands added in Instruction 3 serve as two purposes: (1) They correspond to one of the strands representing state 1, thus they help cell m to transition to state 1 and they partially displace the strand at domain m_3. (2) They serve as a place holder by binding at domains m_4 and m_5 to prevent cell m from being modified in Instructions 4 and 5. Instruction 6 detaches the strand originally bound from domain m_1 to m_3, leaving the domains m_1 and m_2 open. In Instruction 7, the instruction strand attaches to the register at domain m_1 and m_2, which completes the state changing from 0 to 1. □

Claim. The cells to the left of the rightmost state 0 cell stay the same.

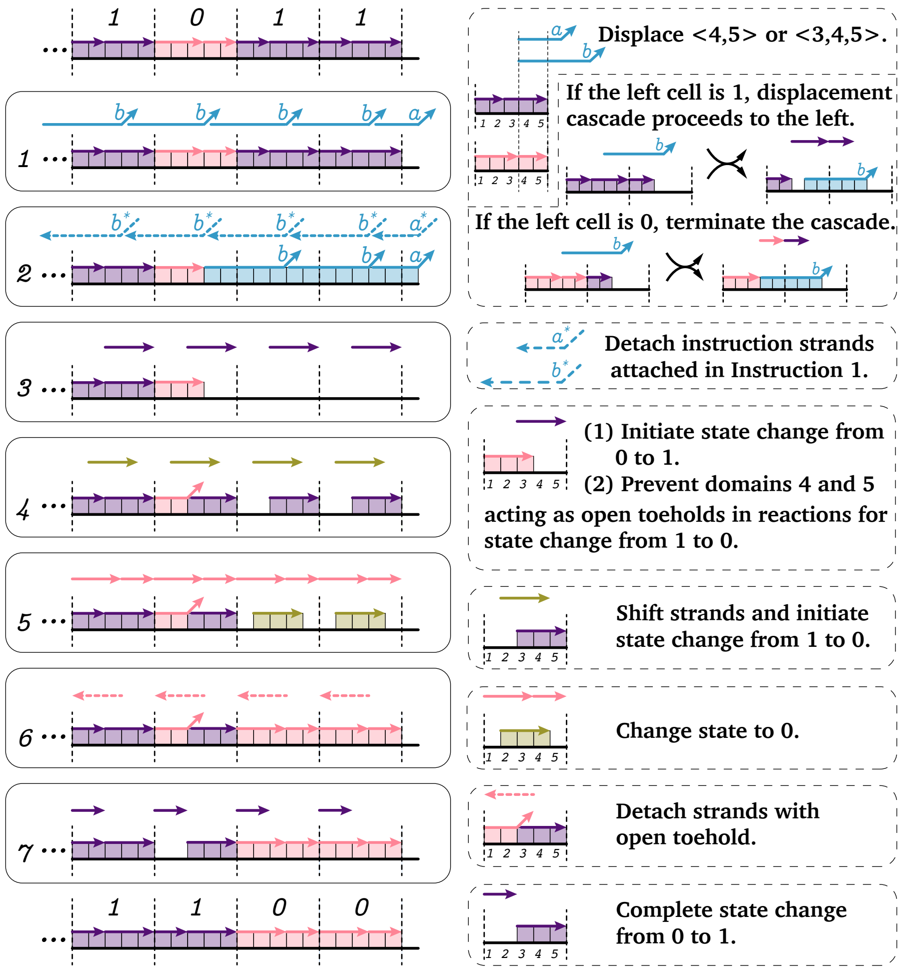
1 0 1 1
Displace <4,5> or <3,4,5>.
If the left cell is 1, displacement cascade proceeds to the left.
If the left cell is 0, terminate the cascade.
Detach instruction strands attached in Instruction 1.
(1) Initiate state change from 0 to 1.
(2) Prevent domains 4 and 5 acting as open toeholds in reactions for state change from 1 to 0.
Shift strands and initiate state change from 1 to 0.
Change state to 0.
Detach strands with open toehold.
Complete state change from 0 to 1.
1 1 0 0

Fig. 4: The program implementing addition by 1 of a binary string on an example register. The top register shows the initial state of each cell. After 7 instructions, the register updates to the state shown at the bottom. Strand colors have three information categories: state 1 (purple), state 0 (pink), intermediates (other colors). Solid boxes show the instruction strands and the state of the register before the strands are applied. Dashed boxes explain the logical meaning of the instructions. The overhang domains a and b are orthogonal to their vertically aligned domains.

Proof. Note that no open toeholds are exposed at cells to the left of cell m, and the displacement cascade does not pass to the left of cell m, thus no changes are made to the states of those cells. □

4 Discussion and future work

4.1 Data preparation

If we do not try to reuse domain sequences, the registers could be prepared by topologically modifying naturally occurring DNA at desired locations through nicking enzymes[6] [20]. If the distance between two nicks is short (for example the length of one domain), the strand in between will spontaneously dissociate, forming a toehold. After the registers with different information are prepared separately and attached to magnetic beads, they are mixed into one solution.

If we reuse domains between cells, the initial preparation of registers requires different techniques. For example, all registers can be initialized to 0 in separate test tubes, and then separate programs executed which move the registers to the desired initial state.

4.2 Experimental feasibility and error handling

Toehold-mediated strand displacement and elution through magnetic beads are well-established techniques, which supports the feasibility of experimental implementation of SIMD||DNA. Other than attaching registers to magnetic beads, registers can also be affixed to the surface of a microfluidic chip. Further, since the instruction strands are added at high concentration and we do not rely on slow mechanisms such as 4-way branch migration, each instruction should finish quickly. However, strand displacement systems can be error prone, and our constructions make several assumptions, the violation of which could lead to various errors.

The first assumption is that waste products from reactions between the instruction strands and registers do not react further with the system. Registers and instruction strands should be allowed to react for a short amount of time before elution such that the high concentration instruction strands interact with the registers, but the low concentration waste products do not. Violating this assumption can cause undesired displacements to occur, leading to possible error in the computation. Interestingly, we conjecture that, besides the reverse of the intended reaction (in the case of toehold exchange), the waste products and registers cannot react in the two programs given here, and therefore our programs are robust to this type of error.

The next assumption is that of a perfect washing procedure where only species with magnetic beads remain after elution. Imperfect washing can result in previous instruction strands reacting with registers undergoing subsequent

[6] For example, Cas9 nickase or restriction enzyme *Pf* Ago, uses an RNA or DNA strand as a guide and can nick at a desired location.

instructions. In practice, the remains of imperfect washing would appear in low concentration so as to have a low probability of affecting the system.

The final assumption is that there is no leak (only toehold-mediated displacements occur). The registers contain nicks where strands could fray and undesired toeholds could open, resulting in strands being mistakenly displaced or incorrect strands binding. Our programs are not robust to leak, raising the question of whether leakless design principles [21–23] can be imposed on the constructions. Leak could also limit the longevity of information stored in our scheme: (toehold-less) four-way branch migration can result in bit exchange errors between different registers. It remains to be seen whether freezing or other means of stabilizing the DNA complexes suffices to ensure long term storage of information encoded in nicked registers.

In addition to preventing errors at the experimental level, it remains open to address errors at the "software level" by employing error correction codes in the data and employing error correction schemes in the instructions.

4.3 Data density

Unlike storing data in the DNA sequence itself, which has a data density of 2 bits per nucleotide, our schemes sacrifice data density. In our schemes, a bit is encoded in a cell, which contains 5 domains. If a domain is composed of 6 consecutive nucleotides, it gives a data density of 0.033 (1/30) bit per nucleotide. It is not obvious that the current construction with 5 domains per cell achieves the highest possible data density for these programs. In practice, there is a tradeoff between the strand binding stability and data density. Here we assume that the minimal number of domains required for two strands to stably bind is two, however in practice the binding strength is affected by experimental buffer (salt concentration) and temperature. Given different experimental conditions, it may be necessary to increase the number of domains in a cell, which could reduce the data density further. However, one gram of DNA can still encode exabytes of information. In principle, data density may also be increased by using different encoding schemes, such as allowing overhangs on the top strands to encode information.

4.4 Uniform versus non-uniform instructions

We can identify instructions as *uniform* or *non-uniform. Uniform* instructions have the property that the same type of instruction strands are added to every cell, as is the case in our programs. *Non-uniform* instructions allow strands to be added to particular cells and not others (e.g., add strands to every seventh cell, or cells 42 and 71). The difference in computational power between uniform and non-uniform instructions remains open, and non-uniform instructions could reduce the number of instructions required for some programs. However, non-uniform instructions could require each cell to be orthogonal in sequence. In contrast, uniform instructions allow every cell to consist of the same sequence, requiring only the domains within the cells to be orthogonal. Sharing the sequences between the cells reduces the number of different instruction strands that need to be synthesized.

4.5 Determinism and nondeterminism

Our programs are designed with *deterministic* instructions: given one state of the register, after adding the instruction strands, the register changes to one specific state. Deterministic instructions make it easy to design, predict, reason about, and compose the programs. In contrast to deterministic instructions, one could also construct *nondeterministic* instructions by introducing nondeterminism to the updates of the cells. For example, consider an empty cell with domains $\langle 3^*, 2^*, 1^* \rangle$, and add instruction strands $\langle 1, 2 \rangle$ and $\langle 2, 3 \rangle$. Either the first or second strand can bind, but since they displace each other, only one will remain after elution. The probability of which strand remains depends on its relative concentration. In principle, applying nondeterministic instructions allows for implementation of randomized algorithms and simulation of nondeterministic computing machines.

4.6 Running time

The running time of a program depends on two factors: running time per instruction and the number of instructions. The running time per instruction depends on whether the instruction updates the cells through *parallel* or *sequential* reactions. In general, instructions are capable of acting on each cell within each register in parallel. Yet, Instruction 1 of the binary counting program does not have this source of parallelism. A first reaction (displacement) must occur on the rightmost cell prior to a second reaction occurring on the second cell, which must occur prior to a third reaction on the third cell, and so on. Thus, this instruction with sequential reactions loses the speedup given by independent instruction reactions occurring in parallel on each cell within a register. Besides the running time per instruction, the larger the number of instructions per program, the more complex is the experimental procedure. This motivates studying the smallest number of instructions required to achieve a computational task.

4.7 Universal computation

Our registers as proposed are restricted to a finite number of cells. So although Rule 110 on an infinite arrangement of cells can simulate an infinite-tape Turing machine, our scheme is only capable of space-bounded computation. To claim that a system is capable of universal computation, it is required that the data tape—in our case, the number of cells—can be extended as needed as computation proceeds. Since our program consists of uniform instructions, domain orthogonality is only required within a cell. Therefore, in principle, the register can be extended indefinitely during computation without exhausting the space of orthogonal domains. The register's length could perhaps be extended by merging bottom strands with top strand "connectors".

4.8 Space-efficient computation

Although Rule 110 is Turing universal, computing functions through simulation of a Turing machine by Rule 110 does not make use of the full power of SIMD||DNA. First of all, while simulation of a Turing machine by Rule 110 was shown to be time-efficient [10], it is not space-efficient. Precisely, simulating a Turing machine on an input which takes T time and $S \leq T$ space requires $p(T)$ time and $p(T)$ space (where $p(T)$ is some polynomial in T). However, Turing machines can be simulated time- and space-efficiently by one-dimensional CA if the automaton is allowed more than two states [19]. Simulating larger classes of CA is a promising approach to space-efficient computation in this model, since our Rule 110 simulation suggests that CA are naturally simulated by SIMD||DNA programs.

4.9 Equalizing encodings

Our two programs use different schemes for encoding binary information in a register. Using some universal encoding would allow applying different consecutive computations to the same registers. Alternatively, we could design programs to inter-convert between different encodings. The reason for suggesting this alternative is that unlike classical machines acting on bits, in SIMD||DNA the way a bit is encoded affects how it can be changed by instruction reactions. For example, in the binary counting program, the encoding ensures that no toeholds except for the rightmost domain are open on the register, which is used to argue correctness. Alternatively, in the Rule 110 program, toeholds must be available throughout the register to achieve the parallel cell updates required by CA. Therefore having one encoding which implements these two different functions seems difficult.

Acknowledgements: The authors were supported by NSF grants CCF-1618895 and CCF-1652824, and DARPA grant W911NF-18-2-0032. We thank Marc Riedel, Olgica Milenkovic, and Tonglin Chen for invaluable discussions. We also thank Marc Riedel for suggesting the analogy to Single Instruction, Multiple Data Computers.

References

1. Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.
2. Donald Beaver. A universal molecular computer. *DNA Based Computers*, 27:29–36, 1995.
3. Dan Boneh, Christopher Dunworth, Richard J Lipton, and Jiri Sgall. On the computational power of DNA. *Discrete Applied Mathematics*, 71(1-3):79–94, 1996.
4. Ravinderjit S Braich, Nickolas Chelyapov, Cliff Johnson, Paul WK Rothemund, and Leonard Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296(5567):499–502, 2002.

5. George M Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science*, 337(6102):1628–1628, 2012.
6. Matthew Cook. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.
7. Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.
8. Rudolf Freund, Lila Kari, and Gh Păun. DNA computing based on splicing: The existence of universal computers. *Theory of Computing Systems*, 32(1):69–112, 1999.
9. Ke Liu, Chao Pan, Alexandre Kuhn, Adrian Pascal Nievergelt, Georg E Fantner, Olgica Milenkovic, and Aleksandra Radenovic. Detecting topological variations of DNA at single-molecule level. *Nature communications*, 10(1):3, 2019.
10. Turlough Neary and Damien Woods. P-completeness of cellular automaton rule 110. In *Automata, Languages and Programming*, pages 132–143, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
11. Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher N Takahashi, Sharon Newman, Hsing-Yeh Parker, Cyrus Rashtchian, Kendall Stewart, Gagan Gupta, Robert Carlson, John Mulligan, Doug Carmean, Georg Seelig, Luis Ceze, and Karin Strauss. Random access in large-scale DNA data storage. *Nature Biotechnology*, 36(3), March 2018.
12. Lulu Qian, David Soloveichik, and Erik Winfree. Efficient turing-universal computation with DNA polymers. In *DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, pages 123–140. Springer, 2010.
13. Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.
14. Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, pages 114–131. Springer, 2014.
15. Paul Wilhelm Karl Rothemund. A DNA and restriction enzyme implementation of Turing Machines. *DNA based computers*, 27:75–119, 1995.
16. Sam Roweis, Erik Winfree, Richard Burgoyne, Nickolas V Chelyapov, Myron F Goodman, Paul WK Rothemund, and Leonard M Adleman. A sticker-based model for DNA computation. *Journal of Computational Biology*, 5(4):615–629, 1998.
17. Dominic Scalise and Rebecca Schulman. Emulating cellular automata in chemical reaction–diffusion networks. *Natural Computing*, 15(2):197–214, 2016.
18. Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.
19. Alvy Ray Smith, III. Simple computation-universal cellular spaces. *J. ACM*, 18(3):339–353, July 1971.
20. S Kasra Tabatabaei, Boya Wang, Nagendra Bala Murali Athreya, Behnam Enghiad, Alvaro Gonzalo Hernandez, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards: Encoding data on native DNA sequences via topological modifications. *bioRxiv*, 10.1101/672394.
21. Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless DNA strand displacement systems. In *DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, pages 133–153. Springer, 2015.
22. Boya Wang, Chris Thachuk, Andrew D Ellington, and David Soloveichik. The design space of strand displacement cascades with toehold-size clamps. In *DNA*

Computing and Molecular Programming, volume 10467 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2017.

23. Boya Wang, Chris Thachuk, Andrew D Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018.
24. Stephen Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55:601–644, Jul 1983.
25. David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103, 2011.