Race Logic: Abusing Hardware Race Conditions to Perform Useful Computation

THIS ARTICLE PROPOSES A COMPUTING APPROACH CALLED RACE LOGIC, IN WHICH INFORMATION IS REPRESENTED AS A TIMING DELAY AND COMPUTATIONS ARE BASED ON THE OBSERVATION OF THE RELATIVE PROPAGATION TIMES OF SIGNALS INJECTED INTO THE CIRCUIT. IN THIS APPROACH, MANY FUNDAMENTAL INFORMATION-PROCESSING OPERATIONS CAN BE EXPRESSED EFFICIENTLY THROUGH THE MANIPULATION OF THE DELAY CHAINING, WHICH RESULTS IN SUPERIOR LATENCY, THROUGHPUT, AND ENERGY EFFICIENCY.

Advait Madhavan
Timothy Sherwood
Dmitri Strukov
University of California, Santa Barbara

As we move toward the end of Dennard scaling, process-technology-based power and performance enhancements are on the decline. Although transistor budgets are ample, architects are now faced with the problem[1] of not being able to power all these transistors simultaneously. This so-called dark silicon problem is pushing architecture into a new era in which application-specific accelerator cores are on the rise and generality is being traded off for significant improvements in performance and energy. Computing devices that we interact with daily, from our phones to our laptops, to our datacenter processors, now carry significant application-targeted hardware functionality. When relaxing the constraints of general purposeness, we begin to see a larger array of solutions opening up. At one end of the spectrum, application customization can rely solely on traditional encoding techniques, as is the case with existing field-programmable gate array (FPGA) supercomputers designed to accelerate problems in bioinformatics.[2] Fully customized circuits are also very common—for example, to speed up audio and image processing.[3] At the other end of the spectrum are somewhat exotic fully customized systems that exploit novel physics and are based on nontraditional technologies such as the D-Wave computer, which uses quantum annealing phenomena to solve optimization problems.[4]

We propose a novel computing approach, called *race logic,* which uses a new data representation to accelerate a broad class of optimization problems, such as those solved by dynamic programming algorithms. The core idea of race logic is to deliberately engineer race conditions in a circuit to perform useful computation. Information is represented as a timing delay, rather than being represented as

Published by the IEEE Computer Society

logic levels (as in conventional logic). Computations can then be based on the observation of the relative propagation times of signals injected into a configurable circuit (that is, the outcome of races through the circuit).

In this approach, the set of arithmetic and logic operations that can be most efficiently expressed changes, leading to new tradeoffs and architectures. Through the manipulation of the natural delay chaining inherent to digital designs, the basic operations of MIN, MAX, and ADD-BY-CONSTANT can be implemented in a way that results in superior latency, throughput, and energy efficiency for certain classes of problems. The big questions are then how is it that these new operations can be implemented and composed, and, perhaps more importantly, is it ever really possible for these compositions to beat a highly optimized traditional design?

With these questions in mind, we implemented, with conventional CMOS technology, a synchronous version of race logic and compared it to a state-of-the-art systolic array implementation. To make the evaluation of this idea more concrete, we examined race logic performance using the example of a well-studied DNA global sequence alignment task, which can be extended to similar graph traversal problems.

Dynamic programming, graph traversal, and sequence alignment

A common problem in bioinformatics is to estimate the similarity between DNA or protein sequences. Needleman and Wunsch's approach may be the easiest to understand[5]; it involves reducing the string similarity problem to either a shortest or longest path problem on a directed acyclic graph (DAG). The notion of similarity between DNA sequences is governed by a metric known as the *edit distance*, which is a measure of the number of insertion, deletion, or substitution operations required to convert one string to another. Each of these edit operations is weighted using a score matrix, such that similarity is rewarded and dissimilarity is punished (for shortest-path formulation), or vice versa (for longest-path formulation). An edit graph is constructed (see Figure 1g) that is a 2D representation of all the possible alignments between the two strings using the score matrix values (see Figure 2) as their edge weights. Any particular alignment is simply a

(a)

$\mathcal{P}$	_	A	C	T	G	_	A	_	G	A
$\mathcal{Q}$	G	A	_	T	_	T	_	C	G	A

(b)

$\mathcal{P}$	0	1	2	3	4	4	5	5	6	7
$\mathcal{Q}$	1	2	2	3	3	4	4	5	6	7

(c)

$\mathcal{S}$	2	3	5	6	8	10	12	14	15	16

(d)

$\mathcal{P}$	A	C	T	G	A	G	A	_	_	_	_	_	_	_
$\mathcal{Q}$	_	_	_	_	_	_	_	G	A	T	T	C	G	A

(e)

$\mathcal{P}$	1	2	3	4	5	6	7	7	7	7	7	7	7	7
$\mathcal{Q}$	0	0	0	0	0	0	0	1	2	3	4	5	6	7

(f)

$\mathcal{S}$	2	4	6	8	10	12	14	16	18	20	22	24	26	28

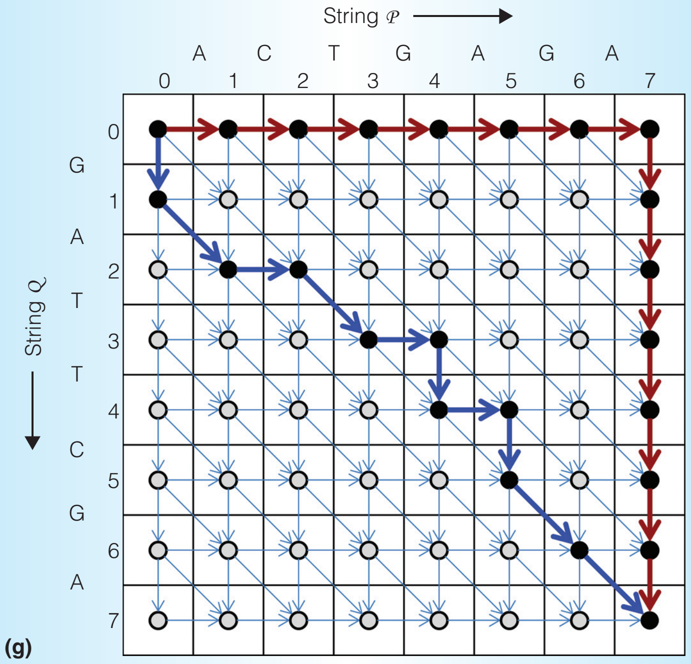


Figure 1. Sequence alignment problem. (a, d) Two possible alignments between strings *P* and *Q* and (b, e) their corresponding alignment matrixes. (c, f) The scores at each node for the particular alignment using the score matrix from Figure 2b. (g) The edit graph.

	A	C	T	G	_
A	1	0	0	0	0
C	0	1	0	0	0
T	0	0	1	0	0
G	0	0	0	1	0
_	0	0	0	0	0

(a)

	A	C	T	G	_
A	1	2	2	2	1
C	2	1	2	2	1
T	2	2	1	2	1
G	2	2	2	1	1
_	1	1	1	1	1

(b)

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \delta(-, P_j) \\ S_{i,j-1} + \delta(Q_i, -) \\ S_{i-1,j-1} + \delta(Q_i, P_j) \end{cases}$$

$$S_{i,j} = \min \begin{cases} S_{i-1,j} + \delta(-, P_j) \\ S_{i,j-1} + \delta(Q_i, -) \\ S_{i-1,j-1} + \delta(Q_i, P_j) \end{cases}$$

(c)

Figure 2. Score matrix. (a) Longest and (b) shortest path score matrixes for the DNA local sequence alignment problem.[6,7] (c) MAX and MIN score functions corresponding to panels (a) and (b), respectively.

path in this graph where every edge corresponds to an edit operation. If similarities were rewarded with a lower edge score than dissimilarities, the shortest path on the graph would yield the best possible alignment. Figures 1a and 1d depict two such alignments, whereas the numbers in each index in Figures 1b and 1e represent the number of symbols encountered until that index. Interestingly, these numbers (also called the *alignment matrix*), when considered as coordinates, represent the nodes on the edit graph that the particular alignment path takes. Figures 1c and 1f show each node's cores for the given alignments, using the score matrix from Figure 2b.

Not only is the edit graph representation a handy tool for visualizing paths and their corresponding alignments, it is also closely tied to the concept of dynamic programming.[6] In particular, dynamic programming relies on solving progressively larger subproblems, starting with a set of small problems and using the results of previous calculations for each new step. Each node on the edit graph calculates the score corresponding to the optimal solution of the subproblem—that is, either the shortest or longest path (depending upon the score matrix)—from the root node to itself. Adjacent nodes use these optimal solutions to calculate their own score as the computation wave proceeds along the diagonal. Because the edit graph itself comprises all possible alignments represented as paths from the root node to the end node, it guarantees searching of the entire space for the most optimal alignment between the given strings.

Although the shortest-path representation is a good way to think about sequence alignment, prior hardware implementations of sequence alignment use systolic arrays or FPGAs to perform the dynamic programming task. The score at any node of a graph is computed from the MIN over the neighbor cells (topologically north, west, and northwest). Because the score is cumulative (and string-length dependent), any application-specific integrated circuit (ASIC) implementation would then need processing elements (PEs) that can store this worst-case cumulative score and lead to large, string-length-dependent sizes. However, the best approaches use clever arithmetic tricks to avoid storing the full precision score matrix and maximally leverage already completed operations. We compared our race logic implementation against a highly optimized systolic array solution by Lipton and Lopresti.[7] They very cleverly mitigated area scaling issues by using maximum-score-dependent modular arithmetic to limit the number of bits of data that must be stored and shared between processing elements and exploit the antidiagonal independence of elements in the edit graph for fine-grained parallelism. Moreover, they also reduced interconnect overhead by developing a tight encoding scheme that interleaves the alphabet and scores.

In contrast to these related works, our method works by representing the edit graph as a set of logical race conditions in a circuit. Because these race conditions are purposely introduced, they are not nondeterministic in the circuit sense. Instead, a pair of input sequences to the system will generate a carefully crafted and controlled set of hardware paths with well-defined timing delays. A rising signal navigates this mesh of timing delays, and the computation performed is directly affected by the result of the differences in the timing of different paths. The signal propagation time affects the final result of the function, and the effect of those timing differences is unknown ahead of time. However, unlike almost every other case where these two properties hold, the way those races are resolved tells us something important about the data applied.

Race logic

In the context of the considered operations on DAGs—that is, the edit graphs—the score at a node is now equivalent to the time it takes for the signal (which is typically injected at the root node) to propagate down the graph to that node in question. We implement this by converting a graph's edge weights to the corresponding timing delays and replacing nodes with either AND or OR gates for MAX and MIN score functions, respectively.

To explain how score functions (Figure 2c) are implemented with race logic, consider the job of one node in the edit graph. It must choose the MIN of multiple different inputs, where each of those inputs is penalized by a constant value. If values are represented by a delay from a reference point *t* (the start of the computation), we can add a constant *c* to a value by simply delaying it by *c* time steps. More concretely, a score of *n* is represented by a Boolean signal 1 appearing at the output of the node *n* unit delays after *t*. Furthermore, when a signal is encoded in time, the MIN operation on a node in the graph receiving multiple inputs is equivalent to passing along the first arriving 1, which can be implemented with a simple OR gate. Similarly, as the AND gate passes the last arriving 1, the AND gate performs the MAX operation. We therefore solve the shortest/longest path DAG problem by measuring the time to propagate the signal from the root node(s) to the output node(s) for a graph, in which all nodes are replaced with OR/AND gates and the edges are replaced with the corresponding delays.

Figure 3a shows an example of a particular DAG, with two input nodes and one output node converted to AND-type (Figure 3b) and OR-type (Figure 3c) race logic circuits.

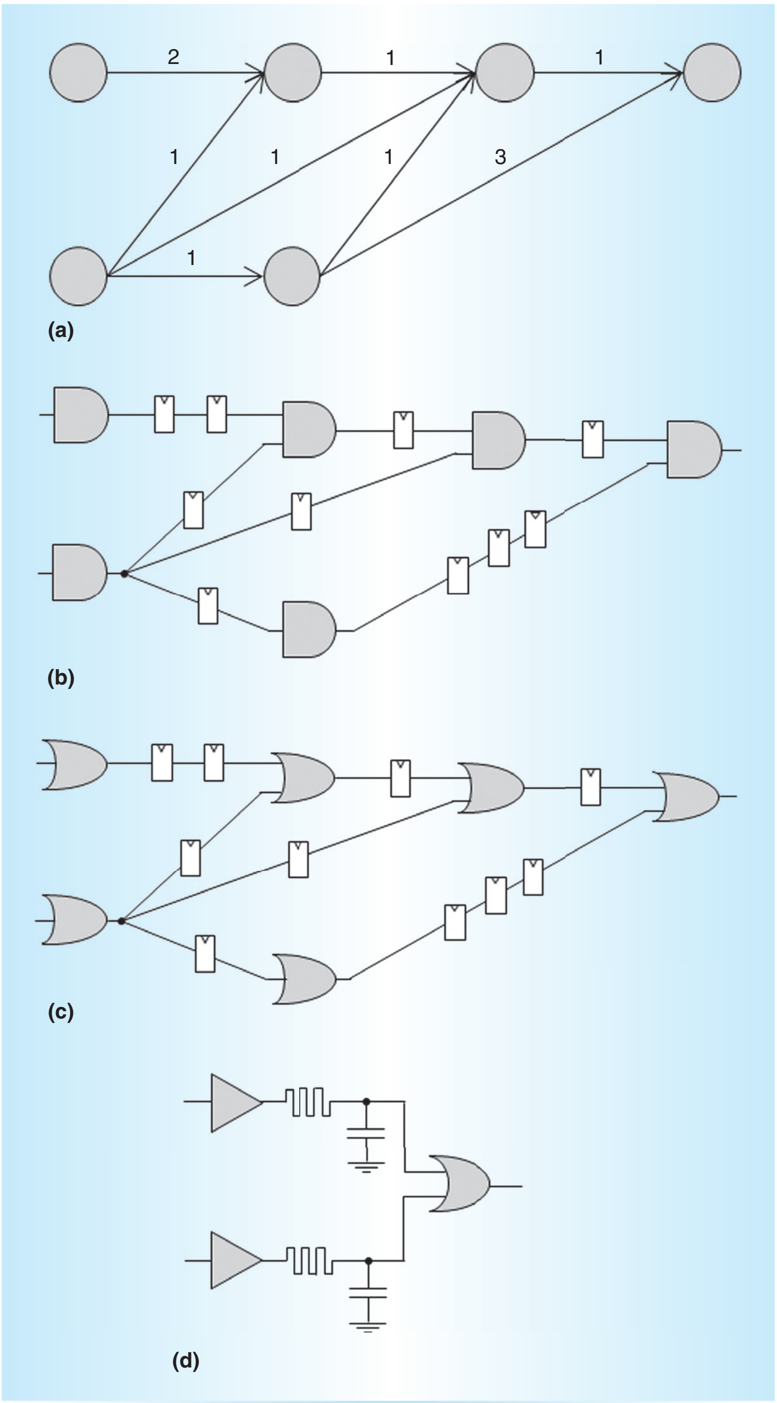
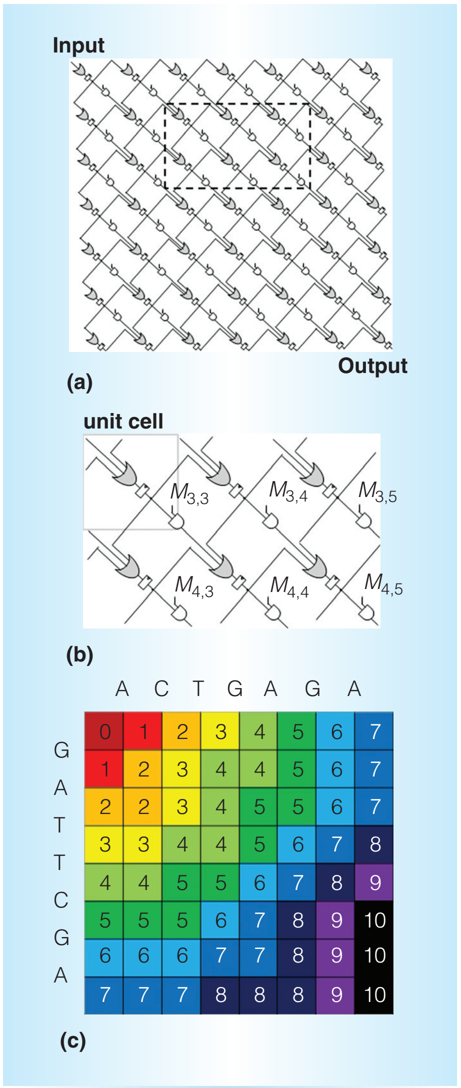


Figure 3. Various race logic implementations. (a) A DAG with weighted edges, and its corresponding synchronous race logic implementation using (b) AND and (c) OR types for longest and shortest path computation, respectively. (d) Example of asynchronous race logic implemented with resistive switching devices.

For synchronous race logic, the unit delay is assumed to be equal to one clock cycle so that *D* flip-flop (DFF) gates implement delay



		A	C	T	G	A	G	A
	0	1	2	3	4	5	6	7
G	1	2	3	4	4	5	6	7
A	2	2	3	4	5	5	6	7
T	3	3	4	4	5	6	7	8
T	4	4	5	5	6	7	8	9
C	5	5	5	6	7	8	9	10
G	6	6	6	7	7	8	9	10
A	7	7	7	8	8	8	9	10

Figure 4. An example of OR-type synchronous race logic implementing global sequence alignment of DNA strings. (a) The circuit for $N = 7$, (b) zoom-in on a particular part, and (c) corresponding example of a signal propagation via race logic for the particular choice of DNA strings shown in Figure 1. The number inside each cell represents timing—that is, the clock cycle at which signal 1 reached the output of an OR gate of a particular unit cell.

elements. In particular, DFFs can be shift-chained when the edge weight is small, or, alternatively, an encoded configuration can be used to implement larger weights. The edit graph can now be thought of as a very deep pipeline with competing paths to the final node from the root node, with all the flip-flops initialized to 0.

To initiate a race computation, both for the OR and AND types of race logic, the input nodes are given a steady value of 1. With every new clock cycle, the 1 signal propagates down the edges of the graph until it reaches another node, where it gets delayed until the other inputs of the node are also 1 in the case of AND-type race logic, or until it just propagates through to the next edge in the case of OR-type race logic. For the specific DAG shown in Figure 3a, it takes two cycles for the 1 signal to propagate to the output node, and we can easily verify that this corresponds to the shortest path. Note that the shortest/longest path value can be converted back to the common representation with a simple counter.

For example, Figure 4 shows an OR-type synchronous race logic implementation of the Needleman-Wunsch algorithm for DNA global sequence alignment with the score matrix from Figure 2b. Here, the signal

$$M_{i,j} = \begin{cases} 1, & Q_i = P_j \\ 0, & Q_i \neq P_j \end{cases}$$

is a matching condition between the corresponding pair of letters of two DNA strings that are being compared, and it is assumed to be implemented with an XNOR gate. As Figure 4 shows, the structure is very uniform and is obtained via the replication of unit cells hosting OR, DFF, and AND gates. In order to simplify the circuitry, we slightly modify the scoring matrix by replacing weights for mismatches from 2 to infinity. It is straightforward to check that the original and modified scoring matrixes are equivalent and thus result in the same score values for the nodes of the edit graph. Figure 4c demonstrates how the signal injected at the input node propagates through the edit graph for particular strings of DNA, which are similar to the previously considered example (Figure 1). It is easy to check that propagation delay corresponds to the best alignment score between these two strings.

Case study

To ensure that the comparison was fair, we implemented the Lipton and Lopresti

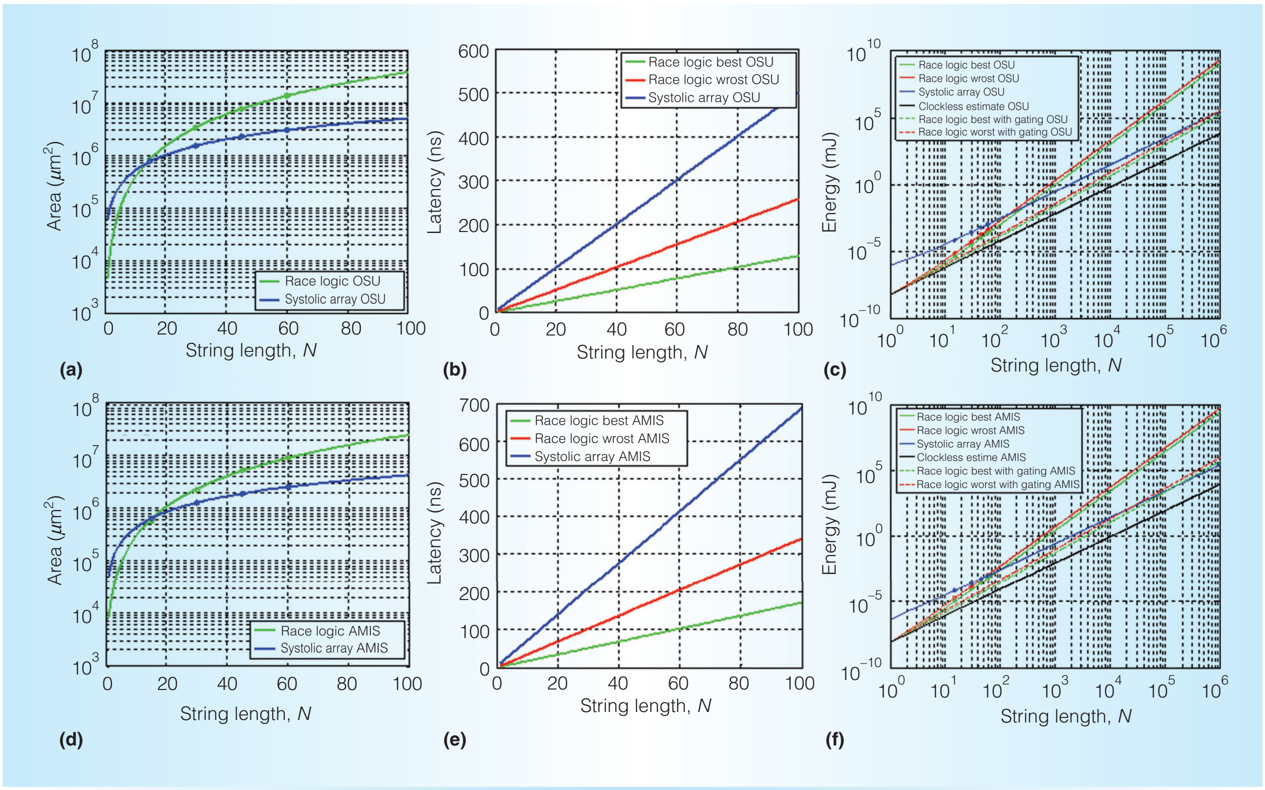


Figure 5. Simulation results. (a, d) Area, (b, e) latency, and (c, f) energy per string comparison operation as a function of string length N for race logic and the Lipton and Lopresti systolic array, for two different CMOS standard cell library implementations. The points on panels a, d, c, and f come from simulation results, whereas solid lines are analytical curves.

architecture using a recent standard cell technology and included all of the area optimizations, score matrices, and encoding schemes that were implemented in the original architecture. We used a half-micron process with multiple standard cell sets—in particular, those from AMI Semiconductor (AMIS) and Oklahoma State University (OSU)—and area numbers were reported from a synthesized Verilog implementation of both architectures. We obtained power and timing information using the Synopsys Primetime tool using a representative set of input vectors. Functional switching activity of each net is generated from these input vectors and is used as toggle information with 100 percent coverage (confidence metric) to estimate power values. Figure 5 shows how the latency, area, energy per computation, and throughput scales with different string length N, using the score matrix shown in Figure 2b.

Results and analytical estimates

Among the simulated performance metrics, area and latency scaling with string length are the easiest to understand. The area of the race logic scales quadratically with N, whereas the systolic array scales linearly. The crossover points in Figures 5a and 5d are due to the simplicity of the race logic unit cell versus that of the complex PE of the systolic array. The latency of both the systolic array and the race logic scales linearly with N, as shown in Figures 5b and 5e. Although the systolic array has a constant latency of $2N - 1$ cycles, the latency of the race logic depends on the degree of alignment, with $N - 1$ and $2N - 1$ cycle latency for perfectly aligned and completely misaligned cases, respectively.

Derivation of energy and power requires more in-depth analysis. Let's assume that C_{clk} corresponds to the capacitances of DFFs that are clocked every cycle, and thus have an activity factor of 1, whereas the $C_{\text{non-clk}}$

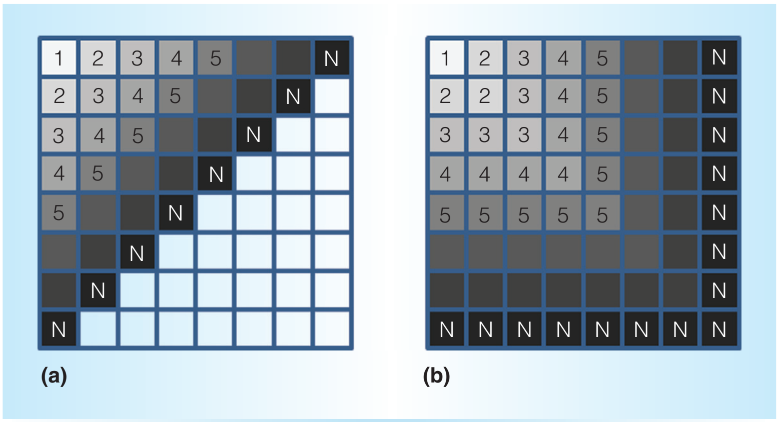


Figure 6. Propagation of wavefront for (a) worst-case and (b) best-case alignment for the score matrix shown in Figure 2b. Each shade of gray shows the wavefront at a different clock cycle as numbered.

corresponds to all other capacitances that have data-dependent activity factors. For both the best-case and worst-case alignment scenarios, all the nonclocked capacitances in the entire architecture are charged once per comparison. You can clearly see this in Figure 6 by following the wavefront. Because the power dissipated is mostly dynamic, it can be written as

$$P = C_{\mathrm{clk}} V_{\mathrm{DD}}^2 N^2 f + C_{\mathrm{non-clk}} V_{\mathrm{DD}}^2 N^2 \alpha f \quad (1)$$

where α is the data-dependent activity factor, V_{DD} is the voltage supply, and f is the frequency of operation. For the systolic array, the power consumption per PE is larger because of increased complexity. However, owing to the linear number of PEs, it scales linearly with array size.

We can calculate the energy consumed per comparison by multiplying power by the time taken per operation. Therefore, energy dissipated per comparison for the best-case and worst-cases alignments are

$$E_{\mathrm{best}} = C_{\mathrm{clk}} V_{\mathrm{DD}}^2 N^3 + (C_{\mathrm{non-clk}} - C_{\mathrm{clk}}) V_{\mathrm{DD}}^2 N^2 \quad (2a)$$

and

$$E_{\mathrm{worst}} = 2C_{\mathrm{clk}} V_{\mathrm{DD}}^2 N^3 + (C_{\mathrm{non-clk}} - 2C_{\mathrm{clk}}) V_{\mathrm{DD}}^2 N^2. \quad (2b)$$

The systolic array implementation with both linear power and latency scaling results in a square-law energy scaling, but with substantially larger constants. Figures 5e and 5f show the crossover points for which the quadratic energy scaling of the systolic array outperforms the cubic scaling of synchronous race logic.

Equations 1, 2a, and 2b define the scaling law of energy and power with respect to N. Because C_{clk} and $C_{\mathrm{non\text{-}clk}}$ cannot be readily estimated, we extract these parameters from fitting experimental data. The resulting equations from fitting for both the AMIS and OSU standard cell libraries are

$$E_{\mathrm{AMIS,best}} = 2.65N^3 + 6.41N^2,$$
$$E_{\mathrm{AMIS,worst}} = 5.30N^3 + 3.76N^2,$$
$$E_{\mathrm{OSU,best}} = 1.05N^3 + 5.91N^2, \text{ and}$$
$$E_{\mathrm{OSU,worst}} = 2.10N^3 + 4.86N^2,$$

where the units of energy are in picojoules.

Energy-optimized architecture

One of the drawbacks of the considered synchronous race logic implementation is its third-order energy scaling with string length N. Equations 2a and 2b reveal that the clocked capacitance is responsible for this cubic behavior because this capacitance, which scales quadratically with string length, is also clocked every cycle. Clock-gating strategies can help alleviate this issue.

The key insight that enables clock gating is that, in race logic, a computation wave flows though the architecture, beginning at the input node and finally arriving at the output. Regions of the fabric that are far from this wavefront are not participating in computation and hence do not need to be clocked. By employing a data-dependent clock-gating strategy, we can turn off unused regions of the chip to save power. Owing to the regular structure of the race logic fabric, we can design the clock network as an H-tree with its granularity as a key parameter.

Consider a 4 × 4 group of cells (multicell region) as shown enclosed in light gray in Figure 7a. We can think of this multicell region as the smallest group of cells that can be gated at once. At a certain time step in the circuit operation, if the gray cells have the Boolean

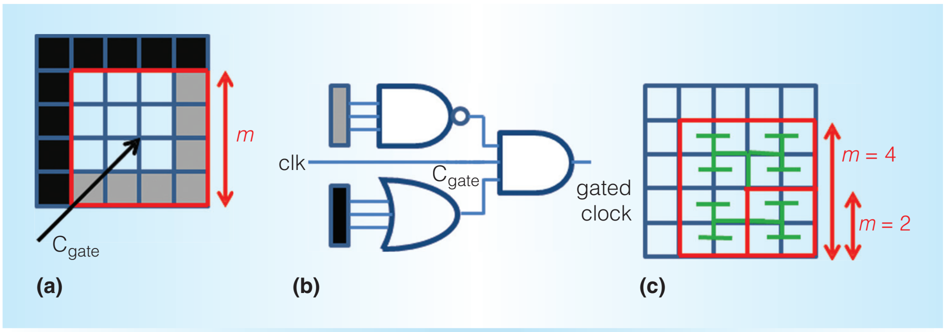


Figure 7. Gating strategies for synchronous race logic. (a) 4 × 4 multicell region with a gated clock, showing associated clock-gating capacitance of C_{gate} and (b) its circuit representation. (c) H-tree-type clock network showing two cases of granularities of gating $m = 2$ and $m = 4$.

value 1, then the wavefront has crossed this multicell region, and their values are not going to change in this operation. Also, if the cells that are in black have the Boolean value 0, it means that the wavefront has not yet approached this multicell region. For both cases, the multicell region shown in Figure 7a doesn't need to be clocked. Deactivating the multicell regions when they are not required helps to significantly reduce energy consumption. Naturally, very fine granularity of a multicell region might not be an optimal choice, because many multicell regions would require clocking every cycle. Alternatively, very coarse granularity could also result in limited energy savings, because one multicell region would end up being clocked for a long time.

To calculate the optimal granularity, we introduce *m,* which is the side length of one multicell region as shown in Figure 7a. Now, the worst-case energy dissipation for the clocked part of the architecture is as follows:

$$E_w = C_{\text{clk}} V_{\text{DD}}^2 N^2 (2m - 2) + C_{\text{gate}} V_{\text{DD}}^2 \frac{N^2}{m^2} (2N - 2),$$

where C_{gate} is the actual capacitance, $(N/m)^2$ is the number of multicell regions, and $2N - 2$ is the total number of cycles. In particular, in Equation 3, the first term represents the entire clocked capacitance being activated only for $2m - 2$ cycles (that is, the worst-case number of clock cycles for which one multicell region remains active), and the second term represents the gating capacitance that the clock distribution network still has to clock. Solving for minimum energy, the optimum granularity is

$$m = \sqrt[3]{\frac{2C_{\text{gate}}(N-1)}{C_{\text{clk}}}}. \qquad (3)$$

By using a novel information representation, computation is simplified and spread out in area, which enables gating strategies to pull energy overheads even lower. Despite this area tradeoff, the throughput per area of the best-case scenario of race logic is considerably better than that of the systolic array for $N < 70$, as Figure 8a shows.

To be clear, we do not believe race logic is any sort of replacement for traditional design practices in general-purpose logic. Rather, it is a new type of data encoding and representation with the opportunity to help improve energy efficiency or speedup for specific information-processing algorithms. Just as multiplication and division operations, which traditionally require complex hardware for binary-encoding schemes, can be simplified to addition and subtraction with logarithmic number systems, a delay encoding transforms other problems, such as MIN-MAX, into a far easier compute-space. Of course, in both examples there will be many other relationships that are then harder to calculate.

At this point, we have proof that some useful computations can be done in this new space, but many open questions remain. What other sorts of computations can be efficiently solved with races? What are the limits

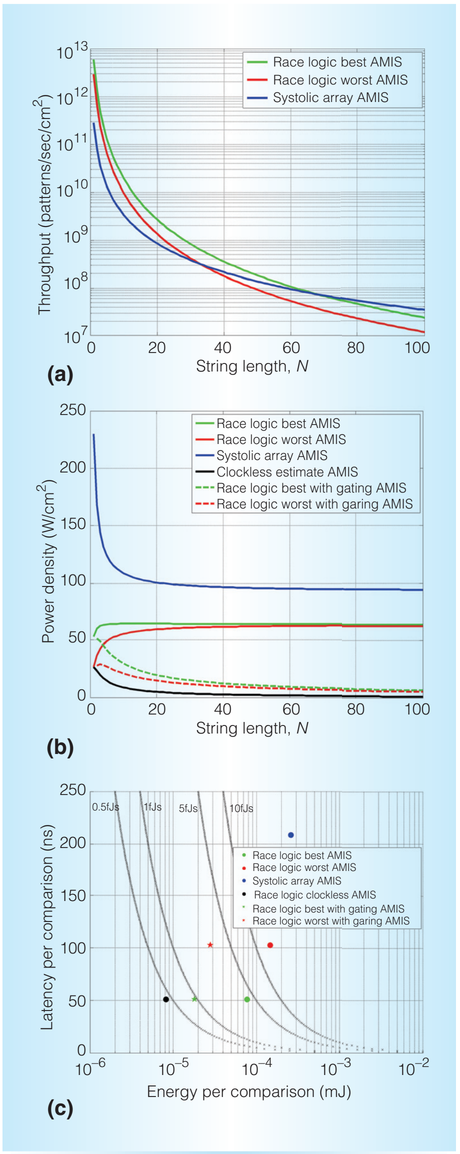


Figure 8. Simulation results. (a) Throughput per unit area and (b) power density as a function of string length *N*. (c) Energy-delay scatter plot for $N = 30$ for race logic and the Lipton and Lopresti systolic array, for the AMIS standard cell library. Black lines on panel *c* represent constant energy-delay curves.

to race logic's expressiveness? Are other timing-based encoding schemes possible? What are the best ways to efficiently implement the expressive and programmable delay elements? Although these questions and many others remain, we at least know now that compositions of the min, max, and add-by-constant primitives provided by this work are sufficient (with the proper routing of values) to solve an important class of bioinformatics similarity problems.

With the field of personalized medicine setting the goal of complete genome sequencing for \$1,000, there has been renewed research interest in, and start-up activity around, this aggressive goal. The ideal machine will need to be fast, small, and affordable—goals that are as much limited by computation as by bioengineering. Any DNA assembly task, whether it is reference-assisted whole genome sequencing or de-novo sequencing, requires the ability to (at very high throughputs) compare strings to one another. Race logic seems to be especially suited for the DNA sequence alignment problem as a similarity threshold is defined, below which strings are assumed to be similar by chance and not due to genuine alignment.[8] This means that in our OR-type race logic implementation, a smaller score can be attributed to a higher level of similarity, and a threshold score can be decided beyond which the architecture will not look for similarity but will move on to the next pattern (a fact that we did not exploit in our results). This means that with an increasing dynamic range, the best-case (rather than worst-case) paths become more representative of a typical situation. Furthermore, the DNA string comparison can be extended to its more complex cousin, protein analysis, which comes with a more complex score matrix. Moreover, even outside the biological realm, a host of applications use dynamic-programming-like solutions, such as dynamic time warping, which is used in speech, and connected digit recognition systems, which could serve as direct applications of this work.

Finally, the most optimal implementation of race logic might be asynchronous and in the analog domain. Most important, the asynchronous race logic does not have a clock network, which is the reason for third-order energy scaling with *N*. This is highlighted by clockless estimates in Figures 5c, 5f, 8b, and 8c. In fact, we can observe that optimized gated design is a step in that direction. Moreover, resistive switching devices can be used to implement configurable edge weights (see Figure 3d),[9] which would provide increased advantages in area and energy.

As our field searches for ways to continue to turn transistors into value without having to toggle those transistors and have them consume power, race logic points in an interesting and little-explored new direction with the potential to significantly outperform traditional designs.

MICRO

Acknowledgments

We thank Alan Mishchenko, Robert Brayton, and Behrooz Parhami for valuable discussions, and the anonymous reviewers for their insightful comments. This work is partly funded by NSF grants CCF-1017579 and CNS-0910734 and the Hellman Family Foundation Fellowship.

References

1. H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling," *Proc. Int'l Symp. Computer Architecture* (ISCA 11), 2011, pp. 365–376.
2. B. Schmidt, *Bioinformatics: High Performance Parallel Computer Architectures*, CRC Press, 2010.
3. K.K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, 1999.
4. M.W. Johnson et al., "Quantum Annealing with Manufactured Spins," *Nature*, vol. 473, no. 7346, 2011, pp. 194–198.
5. S.B. Needleman and C.D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Molecular Biology*, vol. 48, no. 3, 1970, pp. 443–453.
6. N. Jones and P.A Pevzner, *An Introduction to Bioinformatics Algorithms*, MIT Press, 2004.
7. R.J. Lipton and D. Lopresti, "A Systolic Array for Rapid String Comparison," *Proc. Chapel Hill Conf. VLSI*, 1985, pp. 363–376.
8. R. Mott, "Alignment: Statistical Significance," *Encyclopedia of Life Sciences*, 2005, doi:10.1038/npg.els.0005264.
9. J.J. Yang, D.B. Strukov, and D.S. Stewart, "Memristive Devices for Computing," *Nature Nanotechnology*, vol. 8, 2013, pp. 13–24.

Advait Madhavan is a PhD candidate in the Electrical and Computer Engineering Department at the University of California, Santa Barbara. His research focuses on novel methods for information processing, including conceptualization of high-level architectures, analog and digital circuit implementation, and modeling the physics of architectural topologies. Madhavan has an MS in electrical engineering from the University of California, Santa Barbara. He is a member of IEEE. Contact him at advait@ece.ucsb.edu.

Timothy Sherwood is a professor in the Department of Computer Science at the University of California, Santa Barbara. His research focuses on the development of processors exploiting novel technologies, designed for provable properties, and/or implementing hardware-aware algorithms. Sherwood has a PhD in computer science from the University of California, San Diego. He is a senior member of IEEE and an ACM Distinguished Scientist. Contact him at sherwood@cs.ucsb.edu.

Dmitri Strukov is an assistant professor in the Electrical and Computer Engineering Department at the University of California, Santa Barbara. His research focuses on physical implementation of computation, including device physics, circuit design, and high-level architecture, with an emphasis on emerging device technologies. Strukov has a PhD in electrical engineering from Stony Brook University. He is a member of IEEE, the ACM, and the MRS. Contact him at strukov@ece.ucsb.edu.