1

Programmable Parallel Arithmetic in Cellular Automata using a Particle Model

Richard K. Squier, Ken Steiglitz

December 3, 1994

Abstract

In this paper we show how to embed practical computation in one-dimensional cellular automata using a model of computation based on collisions of moving particles. The cellular automata have small neighborhoods, and state spaces which are binary occupancy vectors. They can be fabricated in VLSI, and perhaps also in bulk media which support appropriate particle propagation and collisions. The model uses injected particles to represent both data and processors. Consequently, realizations are highly programmable, and do not have application-specific topology, in contrast with systolic arrays. We describe several practical calculations which can be carried out in a highly parallel way in a single cellular automaton, including addition, subtraction, multiplication, arbitrarily nested combinations of these operations, and finite-impulse-response digital filtering of a signal arriving in a continuous stream. These are all accomplished in time linear in the number of input bits, and with fixed-point arithmetic of arbitrary precision, independent of the hardware.

*Richard Squier is with the Computer Science Department at Georgetown University, Washington DC 20057. Ken Steiglitz is with the Computer Science Department at Princeton University, Princeton NJ 08544.

1 Introduction

Our goal in this paper is to achieve practical computation in a uniform, simple, locally connected, highly parallel architecture — in a way that is also programmable, and thereby accommodates the differing requirements of a variety of applications. Systolic arrays [22], of course, satisfy the locally connected and parallel requirements, but the topology and processor functionality are difficult to modify once the machine is built. In this paper we show how to embed practical computation in one-dimensional cellular automata using a model of computation based on collisions of moving particles [33]. The resulting (fixed) hardware combines the parallelism of systolic arrays with a high degree of programmability.

Fredkin, Toffoli and Margolus [19, 20, 21] have explored the idea of the inelastic billiard-ball model for computation, which is computation-universal. There is also a large literature in lattice-gas automata (see Frisch et. al. [18], for example), which use particle motion and collisions in CA to simulate fluids. We have shown [17] that a general class of lattice-gas automata is computation-universal.

Several recent papers have dealt with particle-like persistent structures in binary CA and their relationship to persistent structures like solitons in nonlinear differential equations [3, 8, 9, 10, 11, 12, 13, 14]. It's fair to say that no one has yet succeeded in using these "naturally occurring" particles to do useful computation. For instance, the line of work in [14] has succeeded in supporting only the simplest, almost trivial computation — binary ripple-carry addition, where the data is presented to the CA with the addend bits interleaved. The reason for the difficulty is that the behavior of the particles is very difficult to control and their properties only vaguely understood.

This paper describes what we think is a promising method of using a particle model and illustrates its application to several different computations. Our basic method will be to introduce the model of computation using particles, and then to realize this model as a cellular automaton. Once the particles and their interactions have been set in the realization, the computation is still highly programmable: the program is determined by the sequence of particles injected. In our examples, we build in enough particles and interactions so that many different computations are possible in the same machine.

CA for the particle model can be realized in conventional VLSI, or — and this is much more speculative — in a bulk physical medium that supports moving persistent structures with the appropriate characteristics [1, 2, 4, 5,

6, 7]. Without distinguishing between these two situations, we'll call the CA or medium a *substrate*. We'll call the substrate, the collection of particles it supports, and their interactions, a *Particle Machine* (PM).

The machines in this paper are one-dimensional, and these PMs are most clearly applicable to computations which can be mapped well to one-dimensional systolic arrays. The computations given as examples in this paper deal with basic binary arithmetic, and applications using arithmetic in regular ways, like digital filtering. Our current ongoing work is aimed toward adding more parallel applications, especially to signal processing, to other numerical computation, and to combinatorial applications like string matching.

Although our main goal is parallelism, it is not hard to show that there are CA of our type that are computation-universal, by the usual stratagem of embedding an arbitrary Turing Machine in the medium and simulating it. We omit details here.

One important advantage of parallel computing using a PM is the homogeneity of the implementation. Once a particular set of particles and their interactions are decided on, only the substrate supporting them needs to be implemented. Since the cells of the supporting substrate are all identical, a great economy of scale becomes possible in the implementation. In a sense, we have moved the particulars of application-specific computer design to the realm of software.

The results in this paper show that PM-based computations inherit the efficiency of systolic arrays, but take place in a less specialized machine. Our examples include arbitrarily nested multiplication/addition and FIR filtering, all using the same cellular automaton realization, in time linear in the number of input bits, and with arbitrary precision fixed-point operands.

This paper has three parts. First we describe the model for parallel computation based on the collisions and interactions of particles. Second, we show how this model can be realized by cellular automata (CA). Finally, we describe linear-time implementations for the following computations:

- binary addition and subtraction;
- binary multiplication;
- arbitrarily nested arithmetic expressions using the operations of addition, subtraction, and multiplication of fixed-point numbers;
- digital filtering of a semi-infinite stream of fixed-point numbers with

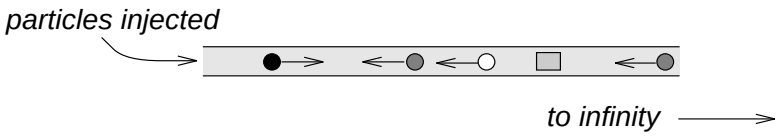

Figure 1: *The general model envisioned for a particle machine.*

a finite-impulse response (FIR) filter (with arbitrary fixed-point coefficients).

2 The Particle Model

We begin with an informal description of the model. Put simply, we want to think of computation as being performed when particles moving in a medium collide and interact in ways determined only by the particles' identities. As we'll see below, this kind of interaction is easy to incorporate into the framework of a CA.

Figure 1 shows the general scheme envisioned for a PM. We think of particles as being injected at different speeds at the left end, colliding along the (arbitrarily long) array where they propagate, and finally producing results. We will not be concerned here with retrieving those results, and for our purposes the computation is complete when the answers appear somewhere in the array. Of course the distance of answers from the input port is no worse than linear in the number of time steps that the computation takes.

In a real implementation, we can either provide taps along the array, or wait for results to come out the other end, however far away that may be. A third alternative is to send in a slowly moving "mirror" particle defined so as to reflect the answer particles back to the input. We can make sure that reflection from the mirror produces particles that pass unharmed through any particles earlier in the array.

We next show how such PMs can be embedded in CA in a natural way. Note that sometimes it's convenient to shift the speed of every particle to change the frame of reference. For example, in the following we'll sometimes assume that some particles are moving left, and others right, whereas in an actual implementation all particles might be moving right at different speeds. The collisions and their results are what matter. This shift in the frame of reference is trivial in the abstract picture we've just given, and will be easy

to compensate for in a CA implementation by shifting the output window of the CA.

3 Cellular automaton implementation

Informally, a CA is an array of cells, each in a state which evolves with time. The state of Cell i at time $t+1$ is determined by the states of cells in a *neighborhood* of Cell i at time t, the neighborhood being defined to be those cells at a distance r (the *radius*) or less of Cell i. Thus, the neighborhood of a CA with radius r contains $k = 2r+1$ cells and includes Cell i itself. When the state space of a cell, $\mathcal{S}$, is binary-valued, that is, when $\mathcal{S} = \{0, 1\}$, we'll call the CA a *binary* CA.

We will escape from the difficulties of using "naturally occurring" particles by expanding the state space of the CA in a way which reflects our picture of a PM. Think of each cell of the CA as containing at any particular time any combination of a given set of n particles. Thus, we can think of the state as an *occupancy vector*, and the state space is therefore now $\mathcal{S} = \{0, 1\}^n$. Implementing such a CA presents no particular problems. We specify the next-state transitions by a table which shows, for each combination of particles that can enter the window, what the next state is for the cell in question. Thus, for a CA with a neighborhood of size k which supports n distinct particles, there are in theory 2^{nk} states of the neighborhood, and hence rows of the transition table. Each row contains the next state, a binary n-vector encoding which particles are now present at the site in question. Only a small fraction of this space is usually needed for implementation, since we are only interested in row entries that correspond to states which can actually occur. This is an important point when we consider the practical implementation of such CA, either in software or hardware.

Figure 2 shows a simple example of two particles traveling in opposite directions colliding and interacting. In this case the right-moving particle annihilates the left-moving particle, and is itself transformed into a right-moving particle of a different type. It is easy to verify that the transition of each cell from the states represented by the particles present can be ensured by an appropriate state-transition table for a neighborhood of radius 1.

Next, we describe some numerical operations that can be performed efficiently in PMs. We start with two simple ways to perform binary addition, and then build up to more complicated examples. Our descriptions will be

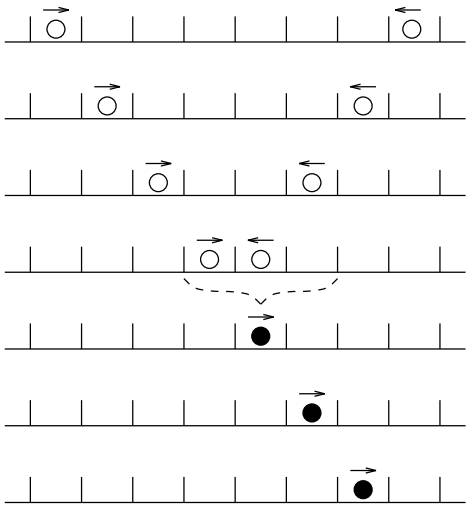

Figure 2: *An example of a collision of a left- and right-moving particle. The left-moving particle is annihilated, and the right-moving particle is transformed into a different kind of right-moving particle. The neighborhood radius in this example is one, as indicated in Row 4.*

more or less informal, but the more complicated ones have been verified by computer simulation. Furthermore, they can all be implemented in a single PM with about 14 particles and a transition table representing about 150 rules. The examples all use a neighborhood of radius 1.

4 Adding binary numbers

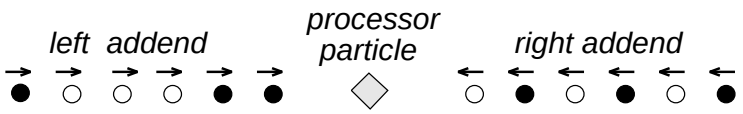

Figure 3: *Implementation of binary addition by having the two addends collide head-on at a single processor particle. There are four different data particles here, left- and right-moving 0's and 1's, represented by unfilled and filled circles. The processor particle is represented by the diamond, and may actually have several states.*

Figure 3 shows one way to add binary numbers. Each of the two addends are represented by a sequence of particles, each particle representing a single bit. Thus there are four particles used to represent data: left- and right-moving 0s and 1s. We'll arrange them with least-significant bit leading, so that when they collide, the bits meet in order of increasing significance. This makes it easy to encode the carry bit in the state of the processor particle.

At the cell where the data particles meet we place an instance of a fifth kind of particle, a stationary "processor" particle, which we'll call p. This processor particle is actually one of two particles, say p_0 and p_1, meant to represent the fact that the current carry bit is either a 0 or a 1. The processor particle starts out as p_0, to represent a 0 carry. The collision table is defined so that the following things happen at a processor particle when there is a right-moving data particle in the cell immediately adjacent to it on the left, and a left-moving data particle in the cell immediately adjacent to it on the right: (1) the two data particles are annihilated; (2) a new, left-moving "answer" particle is ejected; (3) the identity of the processor particle is set to p_0 or p_1 in the obvious way, to reflect the value of the carry bit. Notice that we can use the same left-moving particles to represent the answer bits, as long as we make sure they pass through any right-moving particles they encounter.

We can think of the two versions of the processor particle as two distinct particles, although they are really two "states" of the same particle, in some sense analogous to the ground and excited states of atoms. Thus, we can alternatively think of the different processor particles collectively as a "processor atom", and the p_0 and p_1 particles as different states of the same atom. Either way, the processor atom occupies two slots of the occupancy vector which stores the state. We'll call the particular state of a particle its *excitation state* or, sometimes, its *particle identity*. We'll use similar terminology for the data atom, which has four states representing binary value and direction.

It's not hard to see that this isn't the only way to add, and we'll now describe another, to illustrate some of the tricks we've found useful in designing PMs to do particular tasks. One addition method may be better than another in a given application, because of the placement of the addends, or the desired placement of the sum. The second addition method assumes that the addends are stored on top of one another, in data atoms which are defined to be stationary. That is, the data atoms of one addend are distinct from those of the other, so that they can coexist in the same cell, and their speed is zero (see Fig. 4). We can think of this situation as

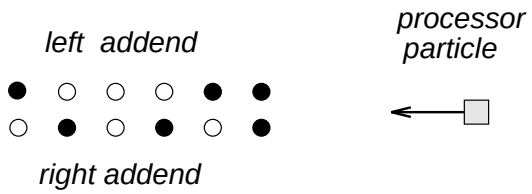

Figure 4: *A second method for adding binary numbers, using a processor atom which sweeps across two stationary addends stored in parallel and takes the carry bit with it in the form of its excitation state.*

storing the two numbers in parallel registers.

We now shoot a processor atom at the two addends, from the least-significant-bit direction. The processor atom simply leaves the sum bit behind, and takes the carry bit with itself in the form of its excitation state. This method reflects the hardware of a ripple-carry adder, and was used in [14].

Negation, and hence subtraction, is quite easy to incorporate into the CA. Just add a particle which complements data bits as it passes through them, and then just add one. (We're assuming two's-complement arithmetic.)

5 Multiplying binary numbers

The preceding description should make it clear that new kinds of particles can always be added to a given CA implementation of a PM, and that the properties we have used, and will use, can be easily incorporated in the next-state table.

Figure 5 shows how a bit-level systolic multiplier [16, 15, 25, 23, 24, 30] can be implemented by a PM. As in the adders, both data and processors are represented by particles. The two sequences of bits representing the multiplicands travel towards each other and collide at a stationary row of processor particles. At each three-way collision involving two data bits and one processor, the processor particle encodes the product bit in its excitation state and adds it to the product bit previously stored by its state. Each right-moving data particle may be accompanied by a carry bit particle. When the data bits have passed by each other, the bits of the product remain encoded in the states of the processor particles, lowest-order bit on the left. Figure 6 shows the output of a simulation program written to verify correct operation

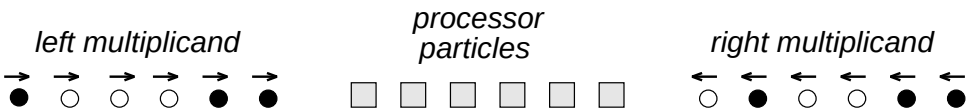

Figure 5: *Two data streams colliding with a processor stream to perform multiplication by bit-level convolution.*

```
R.   .   R.    .    .  p .  p .  p .  p .    .L   .    .L
 .R  .    .R  .    .  p .  p .  p .  p .  L.    .   L.
 .   R.   .   R.   .  p .  p .  p .  p .L   .    .L   .
 .   .R   .    .R  .  p .  p .  p .  pL.    .   L.    .
 .   .    R.   .   R. p .  p .  p .L p .    .L   .    .
 .   .    .R   .    .R p .  p .  pL.  p .   L.   .    .
 .   .    .    R.   .  pR.  p .L p .  p .L   .    .    .
 .   .    .    .R   .  p .R pL.  p .  pL.    .    .    .
 .   .    .    .    R. p .L 1R.  p .L p .    .    .    .
 .   .    .    .     .R pL.  1 .R pL.  p .    .    .    .
 .   .    .    .     .L 1R.  1 .L 1R.  p .    .    .    .
 .   .    .    .   L.  1 .R 1L.  1 .R p .    .    .    .
 .   .    .   .L   .  1 .Lc0R.  1 .  pR.    .    .    .
 .   .    .  L.    .  1L.  0 .Rc1 .  p .R   .    .    .
 .   .   .L   .    .L 1 .  0 . c0R.  p .   R.   .    .
 .   .  L.    .   L.  1 .  0 .  0 .Rcp .    .R   .    .
 .   .L   .   .L   .  1 .  0 .  0 .  1R.    .   R.    .
 .  L.    .  L.    .  1 .  0 .  0 .  1 .R   .    .R   .
.L   .   .L   .    .  1 .  0 .  0 .  1 .   R.   .   R.
```

Figure 6: *CA implementation of PM systolic multiplication, shown with most-significant bit on the right. Row t represents the CA states at time t; that is, time goes from top to bottom. The symbol "R" represents a right-moving 1, "L" a left-moving 1, etc. The computation is* $3 \times 3 = 9$. *The data particles keep going after they interact with the processor particles, and would probably be cleaned up in a practical situation.*

of the algorithm.

6 Nested arithmetic

It is easy enough to "clean up" the operands after addition or multiplication, by sending in particles which annihilate the original data bits. This may require sending in slow annihilating particles first, so one operand overtakes and hits them after the multiplication. Therefore, we can arrange addition and multiplication so the results are in the same form as the inputs. This makes it possible to implement arbitrarily nested combinations of adds and multiplies with the same degree of parallelism as single adds and multiplies.

Figure 7 illustrates the general idea. The product $A \times B$ is added to the product $C \times D$. The sequences of particles representing the operations

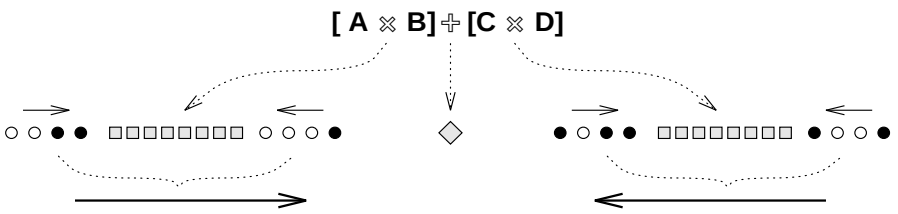

Figure 7: *Illustrating nested computation. Shown is the particle stream injected as input to a particle machine. Collisions among the group on the left will produce the product $A \times B$, with the resulting product moving right, and symmetrically for the group on the right. Then the particles representing the two products will collide at a stationary particle representing an adder-processor, and the two products will be added, as in Fig. 3. The outlined "+" and "×" represent the particle groups that produce addition and multiplication, respectively.*

must be spaced with some care, so that the collisions producing the product finish before the addition begins.

7 Digital filtering

A multiplier-accumulator is implemented in a PM by storing each product "in parallel" with the previously accumulated sum, as in the second addition method, which was illustrated in Fig. 4. The new product and the old sum are then added by sending through a ripple-carry activation particle, which travels along with and immediately follows each coefficient (see Fig. 8). This makes possible the PM implementation of a fixed-point FIR filter by having a left-moving input signal stream hit a right-moving coefficient stream. The multiplications are bit-level systolic, and the filtering convolution is word-level systolic, so the entire calculation mirrors a two-level systolic array multiplier [16, 15, 25, 23, 24, 30]. The details are tedious, but simple examples of such a filter have been simulated and the idea verified.

8 Feedback

So far, we have interpreted the PM substrate as strictly one-dimensional, with particles passing "through" one another if they don't interact. However, we can easily change our point of view and interpret the substrate

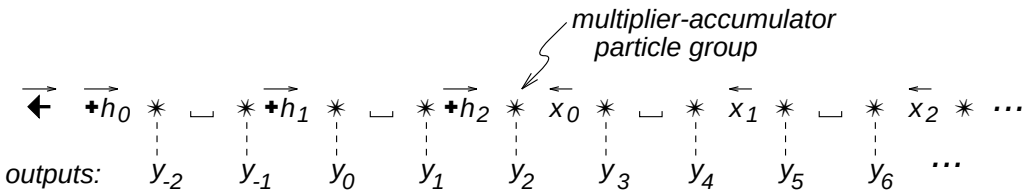

Figure 8: *Particle machine implementation of an FIR filter. The coefficients h_i collide with the input data words x_t at multiplier-accumulator particle groups indicated by "$*$". Each such group consists of a stationary multiplier group which stores its product and the accumulated sum. The ripple-carry adder particles, indicated by "+", accompany each coefficient h_i and activate the addition of the product to the accumulated sum after each product is formed. Finally, on the extreme left, is a right-moving "propeller" particle which travels through the results and propels them leftward, where they leave the machine.*

as having limited extent in a second dimension, and think of particles as traveling on different tracks. We can group the components of the CA state vector and interpret each group as holding particles which share a common track. All tracks operate in parallel, and we can design special particles which cause other particles on different tracks to interact. The entire CA itself can then be thought of as a fixed-width bus, one wire per track.

To illustrate the utility of thinking this way, consider the problem of implementing a digital filter with a feedback loop. Suppose we have an input stream moving to the left. We can send the output stream of a computation back to the right along a parallel track and have it interact with the input stream at the processor particles doing the filter arithmetic. This feedback track requires two bits, one for a "0" and one for a "1" moving to the right. In order to copy the output stream to the feedback track we need an additional particle, which can be thought of as traveling on a third track. This "copying" particle can be grouped with the other processor particles and requires one additional bit in the CA state vector. Altogether we need three additional bits in the state vector to implement feedback processing.

In this multi-track scheme, each extra track enlarges the state space, but adds programming flexibility to the model. Ultimately, it is the size of the collision table in any particular implementation that will determine the number of tracks that are practical. We anticipate that only a few extra tracks will make PMs more flexible and easier to program.

9 Conclusions

We think the PM framework described is interesting from both a practical and a theoretical point of view. For one thing, it invites us to think about computation in new ways, one which has some of the features of systolic arrays, but which is at the same time more flexible and not tied closely to particular hardware configurations. Once a substrate and its particles are defined, specifying streams of particles to inject for particular computations is a programming task, not a hardware design task, although it is strongly constrained by the one-dimensional geometry and the interaction rules we have created. The work on mapping algorithms to systolic arrays [31, 32] may help us find a good language and build an effective compiler.

From a practical point of view, the approach presented here could lead to new kinds of hardware for highly parallel computation, using VLSI implementations of cellular automata. Such CA would be driven by conventional computers to generate the streams of input particles. The CA themselves would be designed to support wide classes of computations.

There are many interesting open research questions concerning the design of particle sets and interactions for PMs. Of course we are especially interested in finding PMs with small or easily implemented collision tables, which are as computationally rich and efficient as possible.

Our current ongoing work is also aimed toward developing more applications, including iterative numerical computations, and combinatorial problems such as string matching [26, 27, 28, 29].

10 Acknowledgement

This work was supported in part by NSF grant MIP-9201484, and a grant from Georgetown University.

References

1. S.A. Smith, R.C. Watt, and S.R. Hameroff, "Cellular automata in cytoskeletal lattices," *Physica* **10D** (1984), pp. 168-174.
2. A.J. Heeger, S. Kivelson, J.R. Schrieffer, and W.-P. Su, "Solitons in conduction polymers," *Rev. Modern Phys.* **60** 3 (1988), pp. 781-850.
3. N. Islam, J.P. Singh, and K. Steiglitz, "Soliton phase shifts in a dissipative lattice," *J. Appl. Phys.* **62** 2 (1987).

4. F.L. Carter, "The molecular device computer: point of departure for large scale cellular automata," *Physica* **10D** (1984), pp. 175-194.
5. J.R. Milch, "Computers based on molecular implementations of cellular automata," presented at *Third Int. Symp. on Molecular Electronic Devices*, Arlington VA, October 7, 1986.
6. S. Lloyd, "A potentially realizable quantum computer," *Science* **261** 17 (Sept. 1993), pp. 1569-1571.
7. C. Halvorson, A. Hays, B. Kraabel, R. Wu, F. Wudl, and A. Heeger, "A 160-femtosecond optical image processor based on a conjugated polymer," *Science* **265** 26 (Aug. 1994), pp. 1215-1216.
8. J. K. Park, K. Steiglitz, and W.P. Thurston, "Soliton-like behavior in automata," *Physica* **19D** (1986), pp. 423-432.
9. C.H. Goldberg, "Parity filter automata," *Complex Systems* **2** (1988), pp. 91-141.
10. A.S. Fokas, E. Papadopoulou, and Y. Saridakis, "Particles in soliton cellular automata," *Complex Systems* **3** (1989), pp. 615-633.
11. A.S. Fokas, E. Papadopoulou, and Y. Saridakis, "Coherent structures in cellular automata," *Physics Letters* **147A** 7 (1990), pp. 369-379.
12. M. Bruschi, P.M. Santini, and O. Ragnisco, "Integrable cellular automata," *Physics Letters* **169A** (1992), pp. 151-160.
13. M.J. Ablowitz, J.M. Keiser, L.A. Takhtajan, "Class of stable multistate time-reversible cellular automata with rich particle content," *Physical Review* **44A** 10 (Nov. 15, 1991), pp. 6909-6912.
14. K. Steiglitz, I. Kamal, and A. Watson, "Embedding computation in one-dimensional automata by phase coding solitons," *IEEE Trans. on Computers* **37** 2 (1988), pp. 138-145.
15. A. Cappello, "A filter tissue," *IEEE Trans. on Computers* **31** (1987), p. 102.
16. J.V. McCanny, J.G. McWhirter, J.B.G. Roberts, D.J. Day, and T.L. Thorp, "Bit level systolic arrays," *Proc. 15th Asilomar Conf. on Circuits, Systems & Computers*, Nov. 1981.
17. R. Squier and K. Steiglitz, "2-d FHP lattice-gasses are computation universal," *Complex Systems*, to appear.
18. U. Frisch, D. d'Humie'res, B. Hasslacher, P. Lallemand, Y. Pomeau, and J. P. Rivet, "Lattice gas hydrodynamics in two and three dimensions," *Complex Systems* **1** (1987), pp. 649-707.
19. C. H. Bennett, "Notes on the history of reversible computation," *IBM J. Res. and Dev.* **32** 1 (Jan. 1988), pp. 16-23.
20. N. Margolus, "Physics-like models of computations," *Physica* **10D**

(1984), pp. 81-95.
21. T. Toffoli, N. Margolus, *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, Cambridge, MA, 1987.
22. H. T. Kung, "Why systolic architectures?" *IEEE Comput.* **15** 1 (Jan. 1982), pp. 37-46.
23. H. T. Kung, L. M. Ruane, D. W. L. Yen, "A Two-level pipelined systolic array for convolutions," in *CMU Conf. on VLSI Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds.), Computer Science Press, Rockville, MD, Oct. 1981, pp. 255-264.
24. S. Y. Kung, *VLSI Array Processors,* Prentice Hall, Englewood Cliffs, NJ, 1988.
25. P. R. Cappello and K. Steiglitz, "Digital signal processing applications of systolic algorithms," in *CMU Conf. on VLSI Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele (eds.), Computer Science Press, Rockville, MD, Oct. 1981, pp. 245-254.
26. H.-H. Liu and K.-S. Fu, "VLSI arrays for minimum-distance classifications," in *VLSI for Pattern Recognition and Image Processing*, K.-S. Fu (ed.), Spinger-Verlag, Berlin, 1984.
27. R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," *1985 Chapel Hill Conference on Very Large Scale Integration*, Henry Fuchs (ed.), Computer Science Press, Rockville, MD, 1985, pp. 363-376.
28. R. J. Lipton and D. Lopresti, "Comparing long strings on a short systolic array," *1986 International Workshop on Systolic Arrays*, University of Oxford, July 2-4, 1986.
29. G. M. Landau and U. Vishkin, "Introducing efficient parallelism into approximate string matching and a new serial algorithm," *ACM STOC*, 1986, pp 220-230.
30. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufman Publishers, San Mateo, CA, 1992.
31. P. R. Cappello and K. Steiglitz, "Unifying VLSI array design with linear transformations of space-time," in *Advances in Computing Research: VLSI Theory*, F. P. Preparata (ed.), JAI Press, Greenwich, Conn., 1984, pp. 23-65
32. D. I. Moldovon and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed systolic arrays," *IEEE Trans. Computers* **C-35** 1 (1986), pp. 1-12.
33. R. K. Squier and K. Steiglitz, "Subatomic particle machines: parallel processing in bulk material," submitted to *Signal Processing Letters*.