Complex Systems **1** (1987) 967–993

Cellular Automata Machines*

Norman Margolus
Tommaso Toffoli
MIT Laboratory for Computer Science,
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract. The advantages of an architecture optimized for cellular automata (CA) simulations are so great that, for large-scale CA experiments, it becomes absurd to use any other kind of computer.

1. Introduction

The focus of the research conducted by the Information Mechanics Group at the MIT Laboratory for Computer Science has been the study of the physical bases of computation, and the computational modeling of physics-like systems. Much of this research has involved reversible models of computation and cellular automata (CA).

In 1981, the frustrating inefficiency of conventional computer architectures for simulating and displaying cellular automata became a serious obstacle to our experimental studies of reversible cellular automata. Even using conventional components, it was clear that several orders of magnitude in performance could be gained by devising hardware which would take advantage of the predictability and locality of the updating process.

The first prototype was a sequential machine which scanned a two-dimensional array of cells, producing new states for the cells fast enough and in the right order so that it could keep up with the beam of an ordinary raster-scan television monitor. After a few years of experimentation and refinement, arrangements were made for a version of our machine, namely CAM-6, to be produced commercially, so that it would be available to the general research community [11,1,12].

The existence of even such small-scale CAMs (cellular automata machines) has already had a direct impact on the subject of CA simulations of fluid mechanics. In informal studies of gas-like models, we found one that Yves Pomeau had previously investigated—the HPP gas [5].[1] According to Pomeau,

*This research was supported by grants from the National Science Foundation (8214312-IST), the Department of Energy (DE-AC02-83ER13082), and International Business Machines (3260).

[1] Pomeau's result was brought to our attention by Gérard Vichniac, then working with our group.

seeing his CA running on our machine made him realize what had been conceived primarily as a *conceptual* model could indeed be turned, by using suitable hardware, into a *computationally accessible* model, and stimulated his interest in finding CA rules which would provide better models of fluids [4].

In fact (as we shall see below), the advantages of an architecture optimized for CA simulations are so great that, for sufficiently large experiments, it becomes absurd to use any other kind of computer.

2. Truly massive computation

Cellular automata constitute a general paradigm for massively parallel computation. In CA, size and speed are decoupled—the speed of an individual cell is not constrained by the total size of the CAM. Maximum size of a CAM is limited not by any essential feature of the architecture, but by economic considerations alone. Cost goes up essentially linearly with the size of the machine, which is indefinitely extendable.

These properties of CAMs arise principally from two factors. First, in conventional computers, the cycle time of the machine is constrained by the finite propagation speed of light—the universal speed limit. The length of signal paths in the computer determines the minimum cycle time, and so there is a conflict between speed and size. In CA, cells only communicate with spatially adjacent neighbors, and so the length of signal paths is inherently independent of the number of cells in the machine. Size and speed are decoupled.

Second, this locality permits a modular architecture: there are no addressing or speed difficulties associated with simply adding on more cells. As you add cells, you also add processors. Whether your module of space contains a separate processor for each cell or time-shares a few processors over many cells is just a technological detail. What is essential is that adding more cells does not increase the time needed to update the entire space—since you always add associated processors at a commensurate rate. For the foreseeable future, there are no practical technological limits on the maximum size of a simulation achievable with a fixed CAM architecture.

The reason that CA can be realized so efficiently in hardware can ultimately be traced back to the fact that they incorporate certain fundamental aspects of physical law, such as locality and parallelism. Thus, the structure of these computations maps naturally onto physical implementations. It is, of course, exactly this same property of being physics-like that makes CA a natural tool for physical modeling (e.g., fluid behavior). Von Neumann-architecture machines emulate the way we consciously think: a single processor that pays attention to one thing at a time. CA emulate the way nature works: local operations happening everywhere at once. For certain physical simulations, this latter approach seems very attractive.

3. A processor in every cell?

In order to maintain the advantages of locality and parallelism, CAMs should be constructed out of modules, each representing a "chunk" of space. The optimal ratio of processors to cells within each module is a compromise dictated by factors such as

1. technological and economic constraints,
2. the relative importance of speed versus simulation size,
3. the complexity and variability of processing at each cell,
4. the importance of three-dimensional simulations,
5. I/O and inter-module communications needs, and
6. a need for analysis capabilities of a less local nature than the updating itself.

Just to give an idea of one extreme at the fine-grained end of the spectrum, consider a machine having a separate processor for each cell, and some simple two-dimensional cellular-automaton rule built in.[2] We estimate that, with integrated-circuit technology, a machine consisting of 10^{12} cells and having an update cycle of 100 pico-seconds for the entire space will be technologically feasible within ten years. If the same order of magnitude of hardware resources contemplated for this CAM (using the same technology) were assembled as a serial computer with a single processor, the machine might require seconds rather than pico-seconds to complete a single updating of all the cells.

There are serious technological problems which must be overcome before three-dimensional machines of this maximally parallel kind will be feasible. The immediate difficulty is that our present electronic technologies are essentially two-dimensional, and massive interconnection of planar arrays (or "sheets") of cells in a third dimension is difficult. In the short term, this problem can be addressed by time-sharing relatively few processors over rather large groups of cells on each sheet; this allows interconnections between sheets to also be time-shared. The architectures of the CAMs built by our group make use of this idea.

A more fundamental problem which will eventually limit the size of CAMs is heat dissipation: heat generation in a truly three-dimensional CAM will be proportional to the number of cells, and thus to the volume of the array, while heat removed must all pass through the surface of this volume. This and other issues concerning the ultimate physical limits of CAMs will be addressed in section 9.

[2]This approach does not necessarily restrict one to a single specific application. There are simple universal rules (see LOGIC in reference 12) which can be used to simulate any other two-dimensional rule in a local manner.

4. An existing CAM

CAM-6 is a cellular automata machine based on the idea that each space-module should have few processors and many cells. In addition to drastically reducing the number of wires needed for interconnecting modules (even in two dimensions), this allows a great deal of flexibility in each processor while still maintaining a good balance between hardware resources devoted to processing and those devoted to the storage of state-variables (i.e., cell states).

Each CAM-6 module contains 256K bits of cell-state information and eight 4K-bit look-up tables which are used as processors. Both cell-state memory and the processors are ordinary memory chips, similar to those found in any personal computer. The rest of the machine consists of a few dozen garden-variety TTL chips, and one other small memory chip used for buffering cell data as it is accessed. All of this fits on a card that plugs into a personal computer (we used an IBM-PC, because of its ubiquity) and gives a performance, in many interesting CA experiments, comparable to that of a CRAY-1.[3]

The architecture which accomplishes this is very simple.

Cell-state memory is organized as 65536 cells in a 256×256 array, with four bits of state in each cell. The cell states are mapped as pixels on a CRT monitor. To achieve this effect, all four bits of a cell are retrieved in parallel (with the array being scanned sequentially in a left-to-right, top-to-bottom order). The timing of this scan is arranged to coincide with the framing format of a normal raster-scan color monitor—cell values are displayed as the electron beam scans across the CRT. Thus, a complete display of the space occurs 60 times per second.

Such a memory-mapped display is very common in personal computers. What we add (see figure 1) is the following: As the data streams out of the memory in a cyclic fashion, we do some buffering (with a pipeline that stretches over a little more than two scan lines) so that all the values in a 3×3 window (rather than a single cell at a time) are available simultaneously. We send the center cell of this window to the color monitor, to produce the display as discussed above. Subsets of the 36 bits of data contained in this window (and certain other relevant signals) are applied to the address lines of look-up tables: the resulting four output bits are inserted back in memory as the new state of the center cell. In essence, the set of neighbor values is used as an index into a table, which contains the appropriate responses for each possible neighborhood case. Even when a new cell state has been computed, the above-mentioned buffering scheme preserves the cell's current state as long as it is needed as a neighbor of some other cell still to be updated, so that every 60th of a second an updating of the entire space is completed exactly as if the transition function had been applied *to all cells in parallel.*

[3]For the simulation of extremely simple CA rules, without any simultaneous analysis or display processing, any computer equipped with raster-op hardware will be able to perform almost as fast as CAM-6, since this CAM is really just a specialized raster-op processor. These computers will not be able to compete as the processing becomes more sophisticated, or as we add more modules to simulate a bigger space without any slowdown.

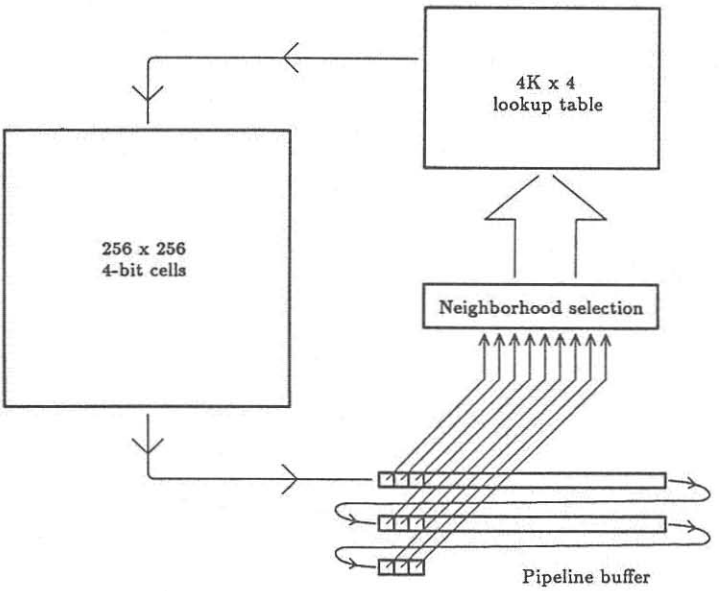

Figure 1: As the four planes are scanned, a stream of four-bit cell values flow through a pipeline buffer. From this buffer, nine cell values at a time are available for use as neighbors. Of these 36 bits, up to 12 are sent to the look-up table, which produces a new four-bit cell value.

Four of the eight available look-up table processors are used simultaneously within each module, each taking care of updating 64K bits of cell-state. The other four *auxiliary* look-up tables can be used, in conjunction with a color-map table and an event-counter, for on-the-fly data analysis and for display transformations. They can also be used directly in cell updating. A variety of neighborhoods are available, each corresponding to a particular set of neighbor bits and other useful signals that can be applied as inputs to the look-up tables. These neighborhoods are achieved by hardware-multiplexing the appropriate signals under software control of the personal-computer host.

Most of CAM-6's power derives from this use of fast RAM tables (which can accomplish a great deal in a single operation) as processors.

Connectors are provided to allow external transition-function hardware (such as larger look-up tables or combinational logic) to be substituted for that provided on the CAM-6 module. Such hardware only needs to compute a function of neighborhood values supplied by CAM-6 and settle on a result within 160 nanoseconds. The CAM-6 module takes care of applying this function to the neighborhood of each cell in turn and storing the result in the appropriate place. If the external source for a new cell-value is a video camera (with appropriate synchronization and A/D conversion), then CAM-6 can be used for real-time video processing.

The connectors also allow external signals to be brought into the module as neighbors, allowing the output of an external random number generator, or signals from other CAM-6 modules, to be used as arguments to the transition function. When several modules are used together, they all run in lockstep, updating corresponding cell positions simultaneously. Three-dimensional simulations can be achieved by having each module handle a two-dimensional slice, and *stacking* the slices by connecting neighbor signals between adjacent slices.

The hardware resources and usage of CAM-6 are discussed in more detail in the book *Cellular Automata Machines: a new environment for modeling* [12]. For illustrative purposes, a few of the physical modeling examples discussed in this book will be surveyed in the next section.

5. Physical modeling with CAM-6

CAM-6 (simply 'CAM' in this section) is a general-purpose cellular automata machine. It is intended as a laboratory for experimentation, a vehicle for communication of results, and a medium for real-time demonstration.

The experiments illustrated in this section were performed with a single CAM module, with no external hardware attached.

Time correlations Figure 2 shows the results of some time-correlation experiments that made use of CAM's event counter [8]. In these simulations, two copies of the same system were run simultaneously, each using half of the machine. Corresponding cells of the two systems were updated at the same moment. Each run was begun by initializing both

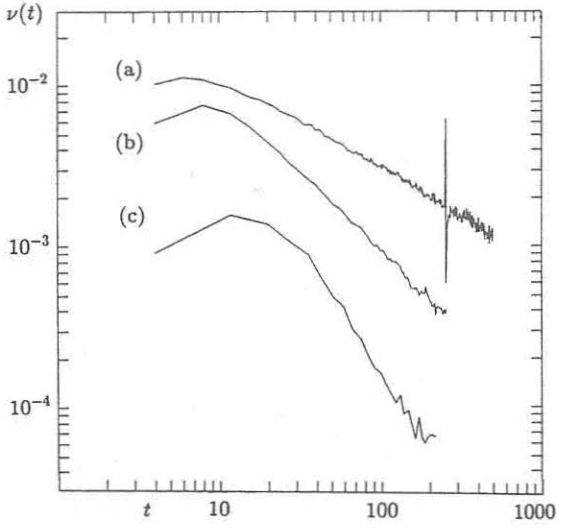

Figure 2: Time-correlation function $\nu(t)$ for (a) HPP-GAS, (b) TM-GAS, and (c) FHP-GAS.

systems with identical cell values, and then holding one of the systems fixed while updating the other a few times. The systems were then updated in parallel for several thousand steps, with a constant time-delay between the two versions of the same system. Velocity-velocity autocorrelations were accumulated by comparing the values of corresponding cells as they were being updated and sending the results of the comparisons to a counter that was read by the host computer between steps. In addition to time-correlations, space and space-time correlations could similarly be accumulated simply by introducing a spatial shift between the two systems before beginning to accumulate correlations. The three time-correlation plots refer to three different lattice gases, HPP [6], TM [11], and FHP [5]; each data point represents the accumulation of over a billion comparisons. The whole experiment entailed accumulating about 3/4 of a trillion comparisons, and took about two and one-half days to run.

Self-diffusion Figure 3 is a histogram showing the probability that a particle of the TM-GAS lattice gas [12] started at the origin of coordinates will be found at a position (x, y) after some fixed number of steps (1024 steps in this case).[4] The data was accumulated by "marking" one of the particles (using a different cell value for it than for the rest, but not changing its dynamics) and then using the auxiliary look-up tables in combination with the event counter to track its collisions, and hence

[4]This experiment was conducted by Andrea Califano.

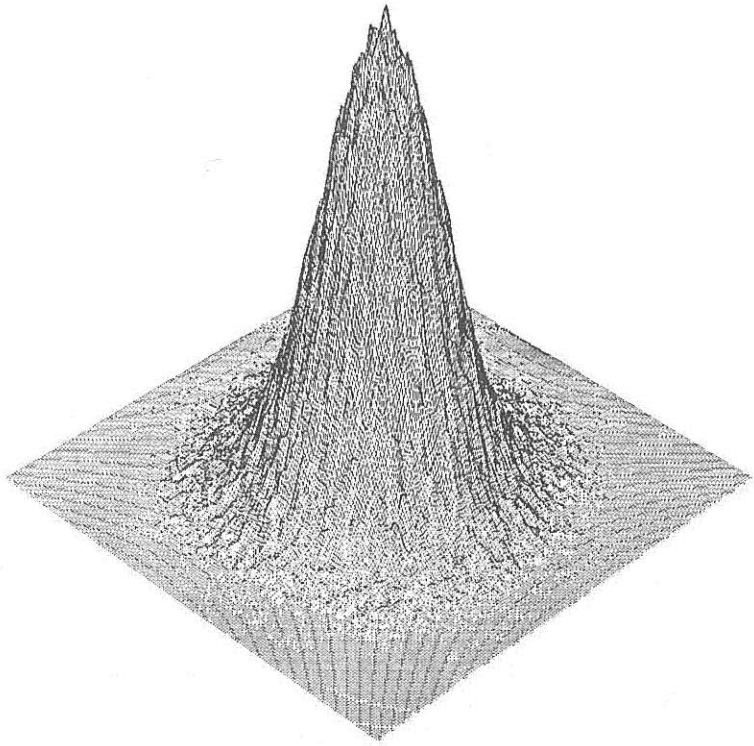

Figure 3: Histogram of $P(x, y; t)$—the probability that a particle of TM-GAS will be found at x, y at time t—as determined by a long series of simulation runs on CAM.

its movements. For each (x, y) value, the height of the plot indicates the number of runs in which the particle ended up at that point.

Though such an experiment requires a massive amount of computation, the essential results of each run can be saved in a condensed form (as a string of collision data for a single particle) for post-analysis. In this way, a single experiment can be used for studying various kinds of correlations.

Thermalization Figure 4 shows the expansion of a clump of particles of TM-GAS. In this experiment, one bit of state within each cell is devoted to indicating whether or not that cell contains a piece of the wall; this bit represents a boundary-condition *parameter* of the simulation, and doesn't change with time. Other state information in each cell is used to simulate the moving gas. Cells which don't border on a wall follow the TM-GAS rule (similar to the better known HPP-GAS rule [5]). Near a wall, the rule is modified so that particles are reflected. An arbitrary

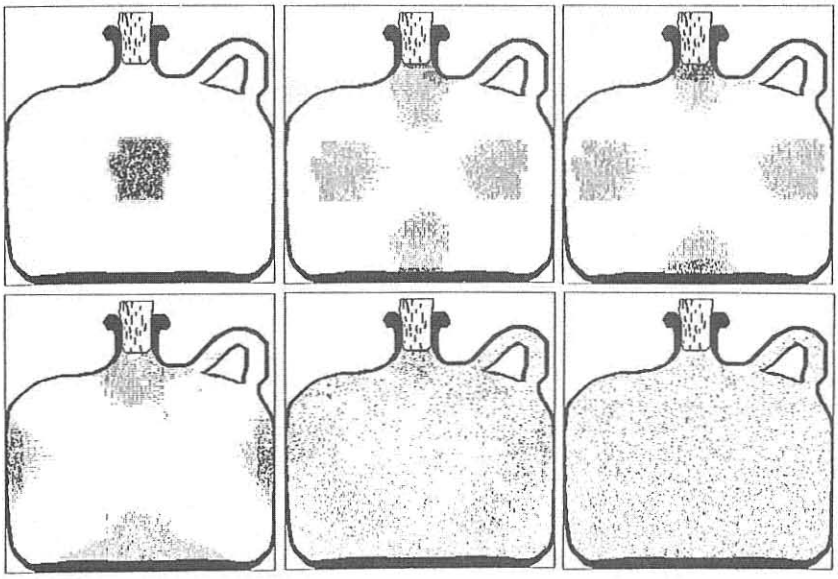

Figure 4: Expansion of a TM-GAS cloud in a vacuum. Repeated collisions between particles and with container's walls eventually lead to thorough thermalization.

boundary can be simulated simply by drawing it; here we've drawn a jug. Initially, it is evident that there are only four directions of travel available to the particles, but as the gas equilibrates, this microscopic detail becomes invisible.

Reflection and refraction Figure 5 shows exactly the same kind of simulation as figure 4, but with a different initial condition. Here we've drawn a wall shaped as a concave mirror, and illustrate reflection of a density enhancement which is initially traveling to the right. For compactness, we use here a special kind of high-density nondissipative wave (a "soliton") that this rule supports (on a slightly larger scale, such phenomena can, of course, be demonstrated with ordinary near-equilibrium "acoustic" waves).

In a similar experiment, figure 6 shows the refraction of a wave by a lens. As before, we draw our obstacle by reserving one bit of each cell's state as a spatial parameter denoting whether the cell is inside or outside the lens. Particles outside the lens follow a lattice-gas rule. Inside the lens, this rule is modified so that particles travel only half as fast as outside. (This is accomplished simply by having the particles

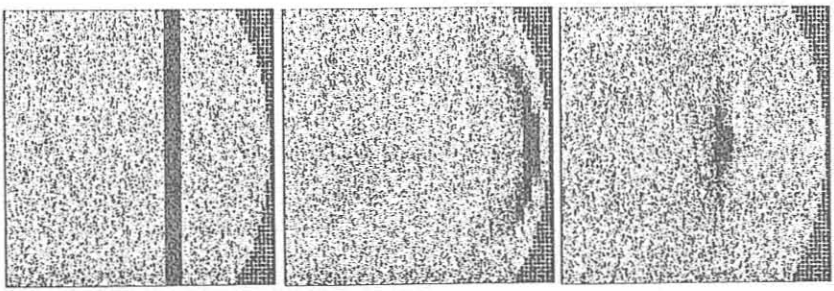

Figure 5: A plane pulse traveling towards a concave mirror is shown (a) right after the reflection and (b) approaching the focal point (c).

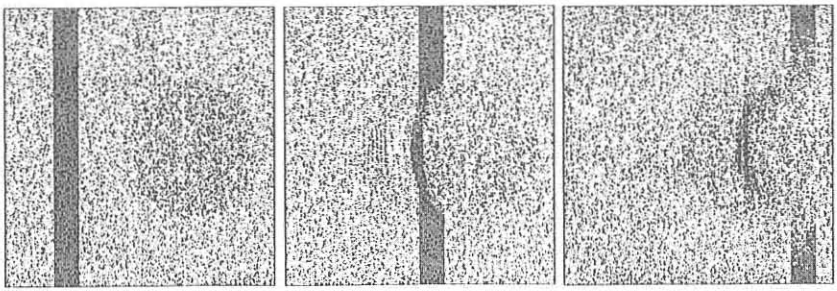

Figure 6: Refraction and reflection patterns produced by a spherical lens.

move only during half of the steps.) Rules that depend on time in such a manner are provided for in CAMs hardware by supplying "pseudo-neighbor" signals that can be seen simultaneously by every cell as part of its neighborhood, and can be changed between steps under software control.

Tracing a flow Figure 7 illustrates an experiment in which smoke is used to trace the flow of a lattice gas. Frame (a) shows a lattice gas with a net drift to the right; this is not evident if we don't color the particles to indicate their velocities. Frame (b) shows the diffusion of particles released from a single point. This source is implemented in the same manner as the mirrors and lenses discussed previously; we mark the cells that are to be sources, and follow a different rule there. The

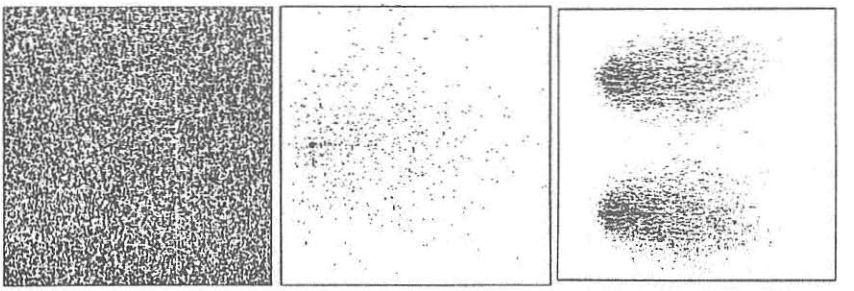

Figure 7: (a) The direction of drift is invisible if the fluid has uniform density. (b) Markers ejected by a smokestack diffuse in the fluid. (c) On a larger-scale simulation, the streamlines start becoming visible.

"smoke" particles released from this source are colored differently from the other particles; however, the dynamics is "color-blind," and treats them just as ordinary gas particles. By looking only at these diffusing smoke particles, one can immediately see their collective net drift. Frame (c) shows the same phenomenon as (b), but using a space 16 times larger (1024×1024 rather than 256×256). Since the width of the diffusion pattern is proportional to $\sqrt{t}$, whereas the net distance a particle drifts is proportional to t, the drift becomes more and more evident as the scale is increased.

The larger cellular automaton shown in that last frame was simulated by a single CAM module,[5] using a technique called *scooping*. The 1024×1024 array of cells resides in the host computer's memory, and CAM's internal 256×256 array is used as a cache: this is loaded with a portion of the larger array, updated for a couple of dozen steps, and then stored back; the process is repeated on the next portion until all of the larger array has been updated. Since scooping entails some overhead (data must be transferred between main memory and cache, and data at the edges of the cache—where some of the neighbors are not visible—must be recomputed in a later scoop), the effective cell-update rate drops somewhat, but to no worse than about half of CAM's normal rate. A similar technique can be used for three-dimensional simulations with a single CAM. (This works particularly well with partitioning rules [7,12].)

Diffusion-limited aggregation Figure 8 shows two stages in the growth of a dendritic structure by a process of diffusion-limited aggregation [14]. There are three coupled systems here, each using one bit of each cell's state. The first system is a lattice gas with a 50 percent density

[5]This experiment was conducted by Tom Cloney.

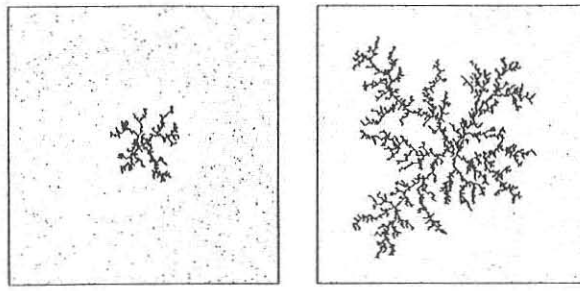

Figure 8: Dendritic growth by diffusion-limited aggregation. The process was started from a one-cell seed in the middle and with a 10 percent density of diffusing particles.

of particles. This gas is used only as a "thermal bath" to drive the diffusion of particles in a second system. The contents of the cells in this second system are randomly permuted in a local manner that depends on the thermal bath. The third system is a growing cluster started from a seed consisting of a single particle: whenever a particle of the diffusing system wanders next to a piece of the cluster, the particle is transferred to the cluster system, where it remains frozen in place. Owing to this capture process, there will be fewer diffusing particles near the growing cluster than away from it, and the net diffusion flow is directed toward the cluster. Most of the new arrivals get caught on the periphery of the cluster, giving rise to a dendritic pattern.

Ising spin systems Figure 9 contains two views of a deterministic Ising dynamics [2,6,9,13]: both frames correspond to a single configuration of spins. The one on the left shows the spins themselves; the one on the right illustrates the use of CAM's auxiliary tables to display in real time a *function* of the system's state rather than the *state* itself—in this case, the bond energy. One can watch the motion of this energy (which is a conserved quantity and thus obeys a continuity equation) while the evolution is taking place; one can run space-time correlation experiments on either magnetization or energy, etc. By using a heat bath (as in the preceding aggregation model), one can also implement canonical Ising models. Figure 10 plots the magnetization in such a model versus the Monte Carlo acceptance probability.[6] Techniques which allow CAM itself to generate (in real time) the finely tunable random numbers needed to implement the wide range of acceptance probabilities used in this experiment are discussed in reference 12. The actual method used

[6]This experiment was conducted by Charles Bennett.

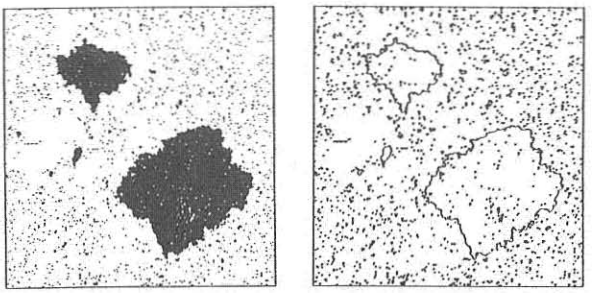

Figure 9: (a) A typical spin configuration; (b) the same configuration, but displaying the energy rather than the spins.

in the experiment plotted here involved using a second CAM machine for this purpose and taking advantage of an instant-shift hardware feature that happens to be present in CAM-6; this feature is central to the design of CAM-7.

Other phenomena Other physical phenomena for which CAM-6 models are provided in reference 12 include nucleation, annealing, erosion, genetic drift, fractality, and spatial reactions analogous to the Zhabotinsky reaction. A number of models which are interesting for the study of the physics of computation are also given, including a reversible cellular-automaton model of computation and some models of asynchronous computation. These examples were developed to illustrate a variety of techniques for using CAM-6; they may also serve to clarify what we mean when we call this device a general-purpose cellular automata machine.

6. CAM-7

If we scale CAM-6 up sixteen-thousandfold, we arrive at a machine with hardware resources comparable to those of a large mainframe computer, but arranged in a manner suitable for extensive scientific investigations using cellular automata. In this and subsequent sections, we will describe our plan for this CAM-7 machine; this design is still undergoing development.

The principal hardware specifications of CAM-7 will be:

1. 2 gigabits of cell-state memory (120-ns dynamic RAM)
2. 1/2 gigabit of look-up-table memory (35-ns static RAM)
3. 8192 plane-modules (each 512×512) operating in parallel

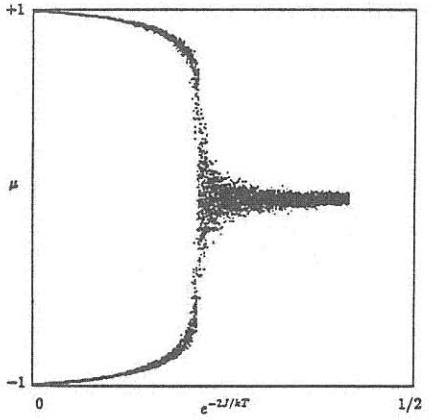

Figure 10: Magnetization μ in the canonical-ensemble model versus the Monte Carlo acceptance probability. Note the sharp transition at the critical temperature T_{crit}.

4. 200 billion cell-bit updates per second (8192 every 40 ns)
5. I/O bus 8192 bits wide, with a 40-ns synchronous word rate (all data appears on this *flywheel bus* once each step)
6. two-dimensional simulations on a $16384 \times 8192 \times 16$ region
7. three-dimensional simulations on a $512 \times 512 \times 512 \times 16$ region
8. any 512×512 region can act as its own TV frame buffer
9. any 16 bits in a 1025×1025 region can be used as a neighborhood

As few as 16 of the plane-modules that constitute a complete CAM-7 machine could be assembled into a $512 \times 512 \times 16$ fractional machine capable of performing 400 million cell-bit updates per second. Such a machine could be integrated into a personal computer much as CAM-6 was, at a similar cost. As many as 100 or more complete CAM-7 machines could be connected together to perform much larger two- or three-dimensional simulations; the constraints are really economic rather than technological.

7. CAM-7 architecture

This machine's speed comes from its parallelism: the machine is made out of ordinary commodity RAM chips, driven at full memory bandwidth, plus some rather simple "glue" logic which will almost all go into a semi-custom

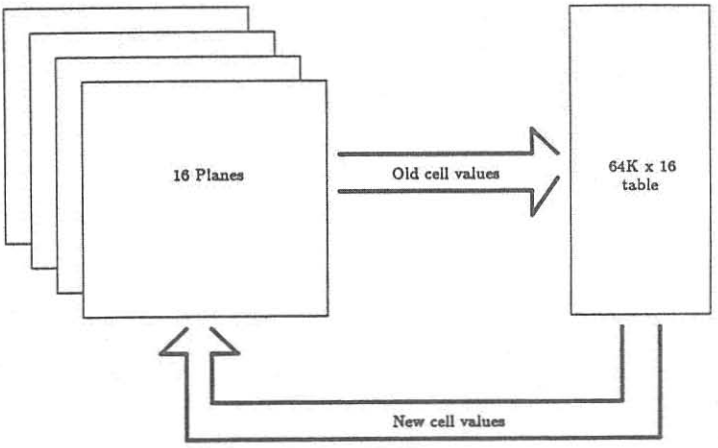

Figure 11: A layer of CAM-7, consisting of 16 plane-modules. As the planes are scanned, a stream of 16-bitcell values are sent as addresses to a 64K × 16 look-up table—the 16-bit results are put back into the planes, as the new cell values.

controller chip associated with each plane-module. We feel that this restriction to inexpensive memory is important, since it should make it economically feasible to build several CAM-7 machines and connect them together to perform CA experiments which involve many trillions of updates per second.

7.1 Basic structural elements

The design really consists of two separate parts: a "data flywheel" which sequentially runs through all the cell data once each step and look-up tables which transform the cell data as it passes through them.

The data flywheel is made up of 8192 plane-modules, each of which is a 512 × 512 × 1 array of bits. The scanning of a module proceeds as for a memory-mapped display (just as it did for CAM-6). Each module puts out one bit every 40 nanoseconds and takes in one bit at the same time.

The look-up tables are each connected to 16 plane-module outputs. Every 40 nanoseconds, they return a set of 16 new cell values which are injected back into the modules (see figure 11).

This selection of module size and update rate is such that the scanning of the modules can be locked to the framing format of a high-resolution monitor, so as to display 512 × 512-pixel images at 60 frames/sec with no interlacing.

When so locked, CAM-7 will update its two gigabits of cell memory 60 times per second. If we decouple the updating from the TV frame rate, CAM-7 will be able to update this entire two-gigabits 100 times per second. When decoupled, we can have each plane-module scan only a fraction of its cells, permitting many more updates per second of this smaller array. For example, if each module scans a region that is only 64×64, then CAM-7 will be able to update a space of size $2048 \times 1024 \times 16$ about 4000 times per second.

7.2 Neighborhoods

The most significant architectural difference between CAM-6 and CAM-7 lies in the way that neighbors are assembled for simultaneous application to a look-up table.

CAM-6 was designed primarily for running CA which employ traditional neighborhood formats, such as the "Moore" and "von Neumann" neighborhoods, in which one cell is updated as a function of more than one cell. Since this machine has many more cells than processors, cells within each module are processed sequentially. Thus, new cell values cannot simply replace old values if the updating is to result in the same state that a simultaneous updating would produce—the old values must be retained as long as they may be needed in computing the new state of some cell. Because of this, CAM-6 requires some buffering of cell values; neighborhood values sent to the look-up table are taken from this buffer (see figure 1). For a 3×3 neighborhood, CAM-6 requires a 515-bit long buffer (2 lines plus 3 bits).

CAM-7 takes as its primary neighborhood format *partitioning cellular automata*, a format wherein space is subdivided into disjoint subsets of cell bits. Lattice gas models follow this format: Each site is updated independently of all the others, and then data is transferred between sites. Since each bit appears as part of only one site, the new values can immediately replace the old ones—no buffering such as was done in CAM-6 is needed. This format has a simpler hardware realization than traditional formats, and allows an enormous range of neighbor choices (as will be explained below).

Thus, a CAM-7 step actually consists of two parts: an updating of all elements of the current partition and a regrouping of data bits to form a new partition. The elements of the partition are just the 16-bit cells, each of which is updated by applying its value to a look-up table and storing the 16-bit result back into the cell. The partition is changed by shuffling bits between cells; how this is done is at the heart of CAM-7's design.

We take advantage of the fact that the plane-module—the elementary "chunk" of CAM-7's space—is much larger than a single cell. The data within one module can be shifted relative to the data in a second module by simply changing the place where we start scanning the data in the first module. Bits are shuffled between cells by shifting entire bit-planes, and this is accomplished by writing to registers that control where the next scan should begin within each plane-module. Since no time is stolen from the updating to accomplish these shifts, we refer to them as "instant shifts." In CAM-7,

neighbors are gathered together by instant shifts.

To avoid complications associated with inter-module communication, consider first how these instant shifts work in a space of size $512 \times 512 \times 16$. Each of the 16 modules consists of one $64K \times 4$ DRAM chip plus a semi-custom controller chip. Given a horizontal and a vertical offset, the controller chip will take care of all of the details; it just has to read the nybbles of the memory chip in an order corresponding to a version of the plane that is shifted (with wraparound) by the given horizontal and vertical offsets. A four-bit pipeline inside the controller chip permits horizontal shifts that aren't a multiple of four. Thus, the 16 bit-planes can be arbitrarily shifted relative to each other between one scan of the space and the next. As each cell is scanned, the 16 bits that come out at a given instant are applied as inputs to a look-up table, and the result is written back to the planes.[7]

The only point remaining to be explained is how the instant-shift process works when the machine is configured so that each bit-plane consists of many plane-modules "glued" together edge-to-edge. What happens is that each module separately performs a shift as described above. The wraparound occurs within each module: cells that should have shifted out the side of one module and into the opposite side of the adjacent module have instead been reinjected into the opposite side of the *same* module. The positions of these cells relative to the edges of a module are exactly as they should be for a true shift: they are just in the wrong module. However, since all modules output corresponding cells at the same moment, each module can produce a truly shifted output by simply replacing its own output with that of a neighboring module when appropriate.

For example, consider CAM-7 running in its $16384 \times 8192 \times 16$ configuration. Each of the 16 bit-planes in this configuration consists of 512 plane-modules, each of which scans an area 512×512. Now suppose we want to shift one of the bit planes 50 positions to the left. Each of the rows within each of the plane-modules is rotated (circularly shifted) 50 positions to the left by appropriately changing the order of accessing the cell memory. Each module's controller chip will produce as an overall output a 512×512 window onto its portion of the complete shifted plane in the following way: The first 462 cell values of each row will come from the plane-module's own rotated data, while the last 50 values will be "borrowed" from the rotated data of the module to its right.

Vertical gluing of bit-planes is achieved in a similar fashion. That is, the controller chip first glues plane-modules together horizontally; the output of this gluing process is further multiplexed across vertically adjacent modules, yielding the final output. In this way, each module only needs to be connected (by a single bidirectional line) to each of its four nearest-neighbor

[7] To save address setup time on the DRAM chips, the controller reads a four-bit nybble from memory and then immediately writes a new value (computed from cells accessed slightly earlier) to that same location. This results in a shift in the physical location of the cells in memory, which is also compensated for by a scan-origin shift within the controller chips.

modules, and any shift of up to 512 positions horizontally, 512 vertically, or any combination of these can be accommodated. Thus, any 16 bits (one from each plane) in a 1025×1025 region can be brought together and used as the neighbors to be jointly sent to the look-up tables.[8]

Of course, if we construct rules where the same table-output value is sent to, say, all 16 planes, then by shifting the planes as described above we can implement not only the traditional neighborhoods but also any other neighborhood entailing up to 16 bits chosen in a 1025×1025 region around each cell. Thus, conventional (i.e., nonpartitioning) cellular automata with very wide neighborhoods can also be simulated on CAM-7, albeit at the cost of using planes and tables rather redundantly.

7.3 Input and output

The basic bus on CAM-7 is the *flywheel bus*, consisting of the final glued outputs of the plane-modules together with inputs to these same modules. The input and output buses are each 8192 bits wide on a full CAM-7 machine: When the machine is operating at its maximum clock rate, a new 8192-bit output word is produced every 40 nanoseconds, and new input words can be accepted at the same rate. Every bit of cell memory in the machine is available to be examined and modified once during every step. External logic (even, if desired, floating-point processors) can be attached here. Depending on how CAM-7 is configured, input bits can be ignored (in favor of internally-generated new cell values), routed as inputs to the look-up tables, or sent directly to the planes.

Besides the two data lines (one for input and one for output) that it contributes to the flywheel bus, each plane-module also has a small number of control lines. Some of these control lines are bussed in bulk to all the modules; the others are merged together into a *control bus* of moderate width. Areas that can be accessed via the control bus include:

1. the look-up table (with auto increment after each read or write)
2. the bit-plane (with auto increment after each read or write)
3. various registers (located within the controller chip)
   (a) the horizontal-offset register
   (b) the vertical-offset register
   (c) the horizontal-size register
   (d) the vertical-size register
   (e) the table-address source select register
   (f) the plane-data source select register
4. various counters (located within the controller chip)

[8] Such large neighborhoods are, for example, particularly useful in image processing.

(a) the address counter

(b) the table-correlation counter

(c) the table-output counter

Each plane-module is connected both to one data line and one address line of a look-up table. During normal updating, the address line is fed sequentially with the glued output of the bit-plane, and the values appearing on the data line are written sequentially into the bit-plane as its new contents. This is, however, just one possible combination of table-address and plane-data sources; by writing to a module's "source select" registers, any of the following may be sent either as an address bit to the look-up table, or as a data bit to be written directly into the plane:

1. the glued output for this plane
2. the output for the plane lying eight positions above or below this one
3. the output from corresponding plane in the other half of the machine
4. the flywheel-bus input for this plane-module
5. one bit from the address counter
6. a constant of zero
7. the complement of any of the above

Notice that the table output doesn't appear in this list—it can only be sent to the plane.

By appropriately controlling the sources both for table addresses and for plane data we can, for example, run a step in which a constant value of 0 or 1 is sent to the table address while the plane data is not affected. The plane can even be shifted during this step, since the table is not needed for this. Thus, one can run steps during which one or more address bits of the table are host-selected constants, analogous to the "phase" bits [12] used by CAM-6. This allows one to split a look-up table into several subtables, to be used during consecutive steps without having to download new tables. Of course, downloading new tables isn't a great problem as long as all the tables are identical (or there are only a few different kinds), since all tables that are the same can be written simultaneously.[9]

Data is read from or written to either planes or tables by the host in a similar manner: A stream of bits is sent to or from the module associated with the data. For planes, the horizontal- and vertical-offset registers are used not only during steps, but also to control where the data-bits sent by

[9]If more flexibility in rewriting tables is needed, a number of microprocessors (say one for every 64 plane-modules) could be added to the design. They could each store a selection of tables, and download them under the command of the host. They could also be useful in generating initial values for the cell states.

the host to the plane should go. For tables, each plane-module controls one bit of the address of a table, and is told by the host which bit of its internal address counter should be shown to its table, to control where data-bits go.

Note that these internal address counters are not provided solely for loading tables; they can also be used during cell-updating, clocked by the 40-ns system clock. By addressing a table with some counter bits, one can, for instance, provide spatial parameters to CA rules (see the rotation algorithm in section 7.8) or perform on-the-fly testing of tables.

7.4 Data analysis

Each plane-module contains a number of counters that are used for real-time data analysis, error detection/correction, or both.

Table outputs are always counted (number of ones in each output). An analysis step can be performed by having some planes remain unchanged (or just shift) while the corresponding table outputs are being counted. For example, if a plane is being used to store a spatial parameter (such as an obstacle in a fluid-flow experiment), the associated table output is not needed for updating and may be programmed for data analysis and counted. If there aren't enough such "free" tables, or if the analysis requires a different neighborhood than the updating, separate analysis steps may be interleaved between updating steps by rewriting tables.

CAM-7 can be operated as two half-machines: Table outputs are continuously compared between corresponding parts of the two halves and the number of differences is counted by the *table correlation counters*. Space and time autocorrelations can be accumulated by running two versions of the same system simultaneously, with a constant space or time shift between them. Since both the number of differences between two corresponding table outputs and the number of ones output by each table separately are counted, the number of occurrences of each of the four possible pairs of binary outputs can be computed. The fact that the sum of the two separate counts plus the correlated count should be even acts as a consistency check for detecting counter errors. If exactly the same system is run in both halves of the machine, the correlation counters detect updating errors.

Note that all counters are double-buffered and can be read at any time by the host without affecting a step that is in progress.

7.5 Error handling

Like CAM-6, each CAM-7 machine will constitute a "building block" from which one can build much larger machines. For example, eight such blocks used together will have two giga-bytes of cell-state memory and will perform one and one-half trillion rather powerful cell-bit updates every second. While there are no inherent architectural limits on how many CAM-7s can be hooked together, there is a practical problem which grows as more and more CAM "blocks" are added, namely error handling. Because of the built-in analysis capabilities described in the previous section, and additional hardware

consistency checks, it will be possible to discover and recover from hardware errors.

Since tables are not supposed to evolve in time, it is relatively straightforward to test whether or not a table contains an error. We can usually detect table errors by performing an analysis step during which all tables are addressed by counter bits. We simply count the number of ones in all table outputs. As long as correlated pairs of tables contain the same rule, we can simultaneously perform a more detailed check by comparing table outputs. Alternatively, we can have the host perform a verify-write of all tables, in which the old contents is read and compared with what the host is writing.

Cell memory is tested by each plane-module during every step. About 22 checksum bits, reflecting the number of ones last written and their positions, are compared to corresponding checksums performed on the data subsequently read. Changing any bit of the configuration will, on the average, change about half of the checksum bits. By dividing all possible 512×512 configurations evenly into more than 10^6 different classes, these checksums make the chance of an undetected plane-memory error very small.

Hard errors, caused by bad components, can be tested for whenever any error is detected. If we run an occasional analysis step during which we test tables, bad chips should always be noticed quickly.

Soft errors, in which memory bits are typically changed, are principally caused by alpha particles. Modern commercial memory chips, which constitute most of CAM-7, are inherently quite reliable: Even with absolutely no provision for error correction, it should be possible to run this machine with 16384 memory chips for several days at a time without any errors. Thus, for a single CAM-7, it may be perfectly practical in most cases to simply detect errors and rerun an experiment if any occur. In fact, for many statistical mechanical experiments, such as fluid flow past obstacles, a rare error in which a bit is dropped doesn't matter at all, so we only need to rewrite incorrect tables and obstacles and watch out for hard errors.

For longer runs, or for large machines built out of many CAM-7s, if we want to guarantee exactly correct operation, it is probably most practical to use each machine as two correlated half-machines, both running the same experiment. Since the chance of two different plane-modules both experiencing a soft error during the same step is extraordinarily small (expected perhaps once in 10^{16} steps for a single CAM-7 machine), we can assume that one out of every correlated pair of plane-modules will always be correct. Planes that were updated incorrectly are fixed by using data from the correct twin, and incorrect tables are simply rewritten. Notice that to correct a plane-module, data doesn't even have to be physically moved from one module to its twin. We can simply run the next step with the correct module providing the input for the tables in both halves of the machine.

Given an error, there remains the problem of deciding which of the pair of correlated plane-modules is incorrect. For plane errors, we rely on the internal checksums maintained by the plane-modules to tell us which module to fix. Otherwise, we make use of one further facility provided by the hardware

in order to quickly and reliably find the error—even if it's a transient one that didn't change the contents of a table. Whenever table comparisons disagree, both the original contents of the cell where the error occurred and the updated value are latched by the controller chips. By examining this information, the host can tell which of the planes was updated incorrectly.

7.6 Three-dimensional operation

When a single CAM-7 is operating in its $512 \times 512 \times 512 \times 16$ configuration, it is, of course, the fact that all plane-modules are updating the same position at the same time that allows information from one layer to be directly available for use by adjacent layers. In terms of plane-modules, one can think of this configuration as being $512 \times 512 \times 8192$, i.e., 8192 deep in the third dimension. We prefer, however, to think of 512 "layers" each consisting 16 consecutive planes, since the outputs from each stack of 16 planes go to common look-up tables.[10]

Each plane-module in this 8K stack is connected to the module eight positions above it and to the one eight positions below it—a total of four wires (input and output above and below) time-shared between all 256K of the cell-bits on each module. Each module has several choices for what it sends as an address to its associated look-up table. It can, of course, send its own glued output. It can also send the glued output of the plane eight positions above or below itself. These three choices make three-dimensional operation straightforward.

For example, each 16-bit cell could be thought of as encoding the contents of a $2 \times 2 \times 2$ cube having two bits at each site. The top eight bits in the cell (i.e., those belonging to the top eight planes in this 16-plane layer) would correspond to the top of the cube, the other eight to the bottom of the cube. After updating the cube according to some rule, let's say that we want to switch to a partition in which the corners of four adjacent cubes become the new cubes. To accomplish this, we will select for each look-up table input the output of the plane-module eight positions above; this is equivalent to shifting all of the plane data eight positions down. Data from the bottoms of one layer of cubes now appear as inputs to the same tables as the tops of the next layer. We must now shift the planes corresponding to the various corners of the old cubes so that the data from four adjacent corners are shifted together. If we've been careful about what order within the cell the results of the first step were placed, we can even use the same rule on these new blocks. If we want different rules on the two partitions, we can of course rewrite the tables before each step. As we alternate between these two partitions, we can avoid a net motion of the cubes by alternately shifting the plane data up and

[10] External logic connected to the flywheel bus inputs and outputs can, of course, group these planes arbitrarily. For example, floating point processors might use them as multi-hundred-bit cells, each containing several floating point numbers that can be separately shifted to change the neighborhood. Since CAM processes each plane-module serially, these floating-point calculations could be pipelined—the delay between starting and finishing processing a cell could be lengthy, as long as a new cell value is completed every 40 ns.

down while moving the blocking back and forth in the other two directions as well.

Just as we could simulate the Moore and von Neumann neighborhoods in two dimensions, we can simulate nearby-neighbor interactions in three dimensions. For example, let's consider a rule that calls for the center cell, its six nearest neighbors, and the center cell in the "past" (i.e., the value the center cell had one step before), with two bits of state for each neighbor. We simply get two bits from the layer above, two from below, and the rest from the current layer (for a total of 16 bits). Our tables should each produce seven two-bit copies of the new value for the center cell, plus one copy of the present value (which will be used as the past by the next step). Four of the copies of the center cell will be shifted one position (north, south, east, and west). One will be visible only to the layer above, one only to the layer below, and the last to the current layer. The look-up tables can now calculate the updated values, and the process can be repeated. Other neighborhoods (for instance, the twelve second-nearest neighbors, or the eight third-nearest neighbors) can all be similarly implemented.

Notice that bits coming from above and below mask the corresponding bits from the current layer. The bit from the current layer no longer appears as an input to this layer's look-up table. You might worry that some bits could become completely hidden and not available as part of the neighborhood of any table, but this is never the case. The masked bit can simply be made visible eight positions down within the current layer, masking another bit which is already visible as part of the neighborhood for the next layer.

What about rules that need more than 16 bits of input? By using some of the bit planes to store intermediate values, rules that need more bits of input can be synthesized as a composition of completely arbitrary 16-input/16-output logical functions. Taking advantage of the strong coupling between the two halves of each CAM-7 machine,[11] one can readily synthesize rather large neighborhoods (up to 32 bits or more) by rule-composition. Note, however, that such compositions can entail, in the worst case, an exponential slow-down as the number of neighbors increases.

7.7 Display

Being able to display the state of our system in real time provides important feedback as to whether or not everything is working as expected, and what parts of the system are doing something interesting that should be investigated more closely.

Two-dimensional display is not much of a problem for CAM-7, since this machine can provide its data in the correct format for a color monitor. This machine can even, if desired, scan its data in the correct format for an interlaced display; since each cell is updated independently of all others, the rows

[11] Recall that any bits from the 16-bit cell in one half can be substituted as table address sources for the corresponding bits in the other half. Machines connected via inputs on the flywheel bus are similarly strongly coupled.

can be scanned in whatever order you choose.

For a complete 16384 × 8192 display, we could cover an enormous wall with 512 color monitors (more if several CAMs are connected), each of which would show a 512 × 512 patch using 64K different colors. Of course, it might be more practical to use only one monitor (or just a few), and shift the data to move the window around. Using interlaced displays and a one-line buffer, 1024 × 1024 or even 2048 × 2048 regions could be viewed on a single monitor.

Since all of the neighbors that would be used for a cell update are available simultaneously, it is a simple matter to display a function of the neighborhood rather than the neighborhood itself. For example, in a fluid-mechanics experiment you might want to show only the smoke particles that trace the flow. Going a step further, part of the machine's resources could be devoted specifically to constructing the image to be displayed. For example, one half of the machine could do the experiment while the other half could monitor the first half, accumulating time-average data for the display.

CAM-7 realizes a three-dimensional system as a stack of two-dimensional layers, each of which can be viewed exactly as discussed above. In its 512 × 512 × 512 × 16 configuration, it would take 512 color monitors to see all layers at once; on the other hand, a single monitor would be enough to see any part of the cube, by shifting the data appropriately (now in three dimensions). Outputs from groups of layers could be combined (e.g., summed, OR'ed, etc.) and shown in a similar manner (still without any external frame buffer). You could even display a sum down through the entire machine—a sort of x-ray.

Suppose we would like to see slices through the cube perpendicular to the plane of our two-dimensional slices. This, and any other 90-degree rotation of the cube about its x, y, or z axis is easily accomplished by CAM using a simple split-and-shift algorithm.[12] Because of the instant shifts available along the bit-planes, rotations about one of the axes can be accomplished in a fraction of a second; rotations about the other two axes would take several seconds.

Such rotations would be particularly useful in conjunction with a display that provides a more natural format for CAMs three-dimensional output.

7.8 A true three-dimensional display

A true three-dimensional display (imagine a translucent cube hanging in mid-air and observable from within a wide angle) is achievable in a relatively straightforward manner. To illustrate the considerations involved, we will describe one particular technique.

Let us first construct a one-bit output for each of CAM-7's 512 layers; in this way, we obtain the equivalent of 512 TV-signal sources, all broadcasting

[12]To rotate a square image, you can first split it into quarters, then shift the four quarters horizontally or vertically until they have each been shifted to a position 90 degrees clockwise of where they started. Each quarter is similarly rotated, and then each eighth, etc., until you reach the level of a single cell. Cells don't look any different when rotated, so you're done.

in parallel. We would like to make up a cube out of these 512 TV frames, by literally stacking them in a third dimension like a deck of cards; as it turns out, it will be expedient to view the resulting "deck" from the top edge rather than from the front side.

Now, construct an array of 512×512 light emitting diodes; each row of LEDs is driven by the outputs of a 512-bit, serial-in, parallel-out shift register with latched outputs (the equivalent of 32 74F673 chips). In turn, the shift registers are fed with the above TV sources, and their outputs latched at the end of every scan line. Thus, the collection of 512 lines produced in parallel by CAM-7's 512 layers will have been captured as a two-dimensional LED picture; this picture, which lies orthogonally to the "cards of the deck," will last about 30 μs before being replaced by the next picture, corresponding to the next scan line.

Every time a new LED picture is ready, we want to display it somewhat below the previous one, so that starting from the top edge of the deck for the first line of the TV frame we will end up at the bottom edge with the frame's last line. This sweeping movement of the LED array is easily achieved by optical means—in a way similar to that demonstrated with success at BBN [10]. That is, the array will be viewed reflected on a thin-membrane mirror stretched over a loudspeaker. The speaker itself will be driven with a 60-Hz sawtooth wave, in sync with CAM-7's internal scan; the resulting slight changes in curvature of the mirror will make the LED array's image sweep through a sequence of focal planes.[13]

Finally, to avoid filling the three-dimensional display with too much data, some selective staining techniques may be appropriate, much as in microscopy. For instance, surfaces can be made visible by simulating "light" within the system: this would consist of particles that travel invisibly in a given direction and light up when they cross a surface (defined by an appropriate local condition).

8. Applications

In addition to statistical mechanical applications that are becoming known (fluid dynamics, Ising spin systems, optics, seismic waves, etc.), CAM-7 should be valuable for a number of less obvious applications.

For example, the structure of CAM-7 seems ideal for certain types of image processing; in particular, for certain "retina-like" tasks where the information contained in detailed two-dimensional images arriving in rapid succession is analyzed and preprocessed in real time by algorithms that are in the main local and uniform, in order to supply a more "brain-like" post-processor with a much smaller amount of pre-digested data.

[13]Note that in the BBN setup, the performance of the system is limited by the available data rate (since the images to be optically multiplexed are generated by drawing vectors on a CRT) rather than by the optical arrangement. CAM-7, on the other hand, has a real-time data rate of 120 gigabits per second, which is more than sufficient to take full advantage of this arrangement.

Each layer could run a different rule, each involving—if desired—rather widely scattered neighbors. Using the three-dimensional connections, with camera input going to the first layer, we could do some consecutive steps of image processing in a pipelined manner: the output of one layer supplying the input for the next.[14] By custom wire-wrapping the flywheel-bus outputs and inputs, a much more complicated pipeline could be achieved. For example, the output of one layer could become the input to several other layers, which could then lead to other layers; there could be further splits and merges, data following a shorter path could be time-correlated with data following a longer path, and so on.

CAM-7 could be used for digital logic simulations in two or three dimensions. Since bit-planes can be made to shift by large amounts between steps, signal speeds would not necessarily be limited to one cell per step. CAM-7 could also be used as a testbed for ideas about using cellular automata VLSI chips as "soft circuitry." For example, given a chip that runs a simple two-dimensional rule such as `LOGIC` [12], one could download a pattern of wires and gates to a chip, and have it simulate the circuit fast enough to actually be used in placed of the target circuit itself.

In general, this machine should be useful in a range of simulation and modeling tasks involving systems which have an appropriate local structure.

9. Conclusions

Although CAM-7, if built, will be by far the fastest computer in the world (for the range of applications for which it is appropriate), it is clearly not pushing the limits of what can be done by cellular automata machines. What are the ultimate limits?

Nothing can simulate a physical system more efficiently than that system itself. Every degree of freedom is fully utilized when a system simulates itself. The reason we simulate a physical system is not one of efficiency; the simulator is better than the original system in some other way. It may be easier to study in detail, or less dangerous, or more versatile, or more accessible, or any number of other things. As we strive to make bigger and faster cellular automata machines, we will ultimately reach a point where it is no longer possible to continue to guarantee exact operation (for example, there is the surface-to-volume problem alluded to in section 3). For a large number of physical simulation tasks, this may not be important; the simulator may still be very valuable anyway. Ultimately, we reach the realm of the universal quantum simulator (see reference 4), which only tries to reproduce the statistics of the results obtained from the original system. Even this might be somewhat loosely considered to be a cellular automata machine, since it will have a finite state at each site and only local interactions. Thus, the question of the ultimate limits of cellular automata machines depends upon

[14] Since each layer can have a different rule stored in its look-up table, CAM-7 as a whole is a true *multiple-program, multiple-data* machine.

where you choose to draw the line—it may well be the same as the question of the ultimate limits of physical simulation.

References

[1] Andrea Califano, Norman Margolus, and Tommaso Toffoli, *CAM-6 User's Guide*; Kenneth Porter, *CAM-6 Hardware Manual*, Systems Concepts, 55 Francisco St., San Francisco 94133 (1987).

[2] Michael Creutz, "Deterministic Ising Dynamics," *Annals of Physics*, **167** (1986) 62–76.

[3] Richard Feynman, "Simulating Physics with Computers," *Int. J. Theor. Phys.*, **21** (1982) 467–488.

[4] Uriel Frisch, Brosl Hasslacher, and Yves Pomeau, "Lattice-Gas Automata for the Navier-Stokes Equation," *Phys. Rev. Lett.*, **56** (1986) 1505–1508.

[5] J. Hardy, O. de Pazzis, and Yves Pomeau, "Molecular Dynamics of a Classical Lattice Gas: Transport Properties and Time Correlation Functions," *Phys. Rev.*, **A13** (1976) 1949–1960.

[6] Hans Herrmann, "Fast Algorithm for the Simulation of Ising Models," Saclay preprint 86-060 (1986).

[7] Norman Margolus, "Physics-like Models of Computation," *Physica*, **10D** (1984) 81–95.

[8] Norman Margolus, Tommaso Toffoli, and Gérard Vichniac, "Cellular-Automata Supercomputers for Fluid Dynamics Modeling," *Phys. Rev. Lett.*, **56** (1986) 1694–1696.

[9] Yves Pomeau, "Invariant in Cellular Automata," *J. Phys.* **A17** (1984) L415–L418.

[10] Lawrence Sher and C. D. Barry, "The Use of an Oscillating Mirror for 3-Dimensional Display," in *New Methodologies in the Study of Protein Configuration*, T. T. Wu, ed. (Van Nostrand, 1985) Chapter 6.

[11] Tommaso Toffoli, "CAM: A High-performance Cellular-automaton Machine," *Physica*, **10D** (1984) 195–204.

[12] Tommaso Toffoli and Norman Margolus, *Cellular Automata Machines—A New Environment for Modeling*, (MIT Press, 1987).

[13] Gérard Vichniac, "Simulating Physics with Cellular Automata," *Physica*, **10D** (1984) 96–115.

[14] Thomas Witten and Leonard Sander, *Phys. Rev. Lett.*, **47** (1981) 1400.