From: Proceedings of the 27th annual international symposium on Computer architecture (ISCA '00)

An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations

Norman Margolus*
MIT AI Laboratory
545 Technology Square
Cambridge MA 02139
nhm@mit.edu

ABSTRACT

Spatial-lattice computations with finite-range interactions are an important class of easily parallelized computations. This class includes many simple and direct algorithms for physical simulation, virtual-reality simulation, agent-based modeling, logic simulation, 2D and 3D image processing and rendering, and other volumetric data processing tasks. The range of applicability of such algorithms is completely dependant upon the lattice-sizes and processing speeds that are computationally feasible. Using embedded DRAM and a new technique for organizing SIMD memory and communications we can efficiently utilize 1Tbit/sec of sustained memory bandwidth in each chip in an indefinitely scalable array of chips. This allows a 10,000-fold speedup per memory chip for these algorithms compared to the CAM-8 lattice gas computer, and is about one million times faster per memory chip for these calculations than a CM-2.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors—*array and vector processors, SIMD*; C.1.4 [**Processor Architectures**]: Parallel Architectures—*distributed architectures*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

Keywords

Virtual Processor, PIM, lattice gas, cellular automata

1. INTRODUCTION

Beginning in 1986, a new class of physical simulation methods were developed based on discrete particles which hop around and collide on a spatial lattice[11, 20]. These methods provide a simplified molecular dynamics that makes direct large-scale simulation of complex fluids and materials computationally feasible[2]. Stimulated by the promise of these new and "embarassingly parallel" *lattice gas* techniques, special purpose SIMD machines were developed that were optimized for these kinds of calculations, including the CAM-8 machine, completed in 1992[17]. Although CAM-8's mesh architecture is indefinitely scalable, only small-scale prototypes were actually built. With an aggregate memory bandwidth a factor of 100 less than a contemporary CM-2 and just 8 lookup-table processors, CAM-8 prototypes ran lattice-gas simulations at about the same speed as the CM-2[7, 17]. This demonstrated the usefulness of optimizing a SIMD architecture for discrete spatial lattice simulations.

With its simple high-level lattice-gas programming model, CAM-8 was used to develop a variety of algorithms for physical simulation, bit-mapped virtual-reality, image processing, and logic simulation[17, 19]. Although CAM-8 applications have focused on novel non-numerical computations, there are also of course many well-known numerical techniques (such as finite-difference and lattice-Boltzmann) that are based on local lattice processing. Moreover, hybrid techniques that combine numerical and symbolic processing are becoming increasingly attractive[3].

The limiting factor in CAM-8 was DRAM memory bandwidth. The lattice-update rate was limited by how quickly the entire simulation state could be read out of an array of memory chips, passed through a set of pipelined processors, and returned to memory. Using embedded DRAM, it is currently possible to sustain about 1Tbit/sec of memory bandwidth per chip using 20 4Mbit blocks of DRAM (filling about half of a 256 mm^2 die with memory). This represents a 20,000-fold increase in memory bandwidth, compared to the memory chips that were used in CAM-8. Mesh-connected arrays of such chips, each handling an equal-sized *sector* of a large spatial simulation (an approach pioneered in the DAP[23]) could make very large and fast lattice applications possible (Figure 1). At such data rates, however, it becomes challenging to arrange for the processing, communications and control to all keep up with the memory. We would also like to do this while retaining a simple and powerful high level programming model.

In this paper we discuss a new memory and communications organization optimized for lattice-oriented computa-

*Permanent address: Boston University Center for Computational Science, 3 Cummington Street, Boston 02215

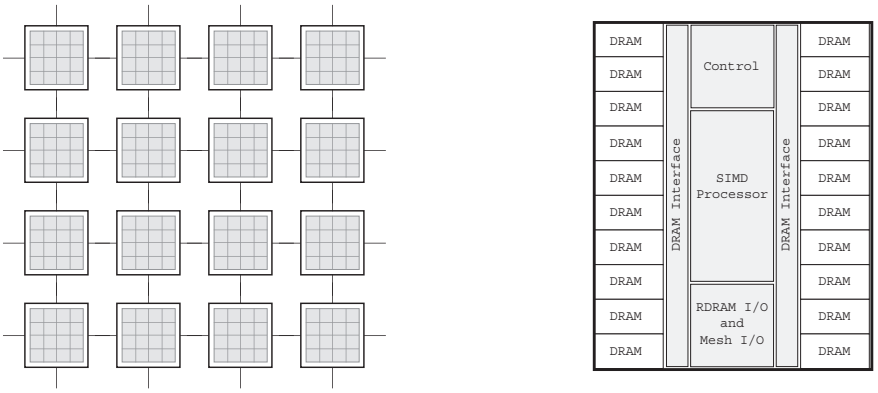

Figure 1: A virtual lattice SIMD computer. (a) An n-dimensional volume of data is divided up evenly among a 1D, 2D or 3D mesh array of processor chips. (b) Each processor chip contains a set of 20 DRAM modules that are accessed in parallel. All 20 I/O words are applied to a set of SIMD processing elements in parallel.

tions that achieves these ends. By taking advantage of translational and rescaling symmetries in mapping the spatial lattice into the granular structure of the memory (words, rows, etc.), essentially perfect use can be made of both memory and communications bandwidth to spatially shift data within the emulated lattice. This scheme is very different from earlier memory optimization schemes used on SIMD machines[27, 13], which took no account of the granular structure of memories. While earlier SIMD schemes have used virtual processors[25], none have taken advantage of the possibility of skewing the locations where the various processors are operating in the emulated lattice, to even out offchip communications needs.

In our lattice hardware, we handle data movement and data processing separately. Data movement is handled by organizing the memory and communication resources of an array of DRAM chips into an n-dimensional *lattice memory*, allowing the processors on each chip to directly access blocks of uniformly shifted data. Spatial shifts incorporated into each memory access can be large, and move lattice data in an arbitrary direction. Many processor architectures are compatible with such a lattice data movement scheme. Using simple few-bit SIMD operations would give a PIM-like array[12], but with lattice data movement added. Alternatively, reconfigurable logic could be added to the DRAM, following (for example) the Active Pages model[22]). We could even use rather conventional vector processors, as the IRAM project does[14], operating the processors in lockstep when performing SIMD operations.

Although simple few-bit SIMD operations are adequate for some applications, the use of moderately sized lookup tables as symbolic processors can speed up the kinds of lattice calculations we're interested in by two to three orders of magnitude, based on our CAM-8/CM-2 comparisons. This approach is as flexible in symbolic lattice computations as using more complex processors, and can take good advantage of the uniformity of our lattice applications. For numerical computations, our scheme for dealing with memory granularity also lets us take advantage of depth-first virtualization: we completely apply a numerical operation to one subset of the lattice data before moving on to another, thus avoiding the need for large numbers of hardware registers to make arithmetic operations efficient[9].

In the next Section, we discuss some novel non-numerical and semi-numerical lattice-based computations developed on CAM-8, as examples of some of the kinds of applications we are targeting. We then describe a novel memory organization technique which allows such computations to map efficiently onto mesh arrays of embedded DRAM chips with SIMD processors. We discuss this technique first in a pure software context, as an algorithm which could be implemented on a conventional microprocessor. We then discuss a hardware realization using embedded DRAM, including a brief discussion of numerical SIMD hardware.

2. LATTICE COMPUTATIONS

A lattice system can be thought of as a kind of n-dimensional bit-map. Computing using such bit-maps permits models to have enormous resolution and detail, but requires concommitant processing and storage. For physical simulation, simple algorithms with a small number of bits at each lattice site are particularly attractive, since they reduce both the storage and processing requirements, making large simulation sizes practical.

In this section we illustrate some simple physical simulations that were implemented on a CAM-8 machine with 128 DRAMs. A single embedded DRAM chip of the sort pictured in Figure 1 would run these algorithms about 100 times faster than this machine—the speedup factor comes entirely from increased memory bandwidth. As pictured in the figure, large arrays of these chips can be used together, to give an additional speedup factor that is proportional to the size of the array.

Figure 2 shows two hydrodynamic simulations using lattice gases. The first is a semi-numerical lattice gas, which uses two-bit integer particle-counts moving on a hexagonal lattice, and interacting at lattice sites[4]. Particles are introduced on the left with a rightward momentum, flow past the half-cylinder, and exit on the right. A second tracer gas (also introduced on the left) follows the flow, and is used to produce the streamlines shown. The simulation illustrates

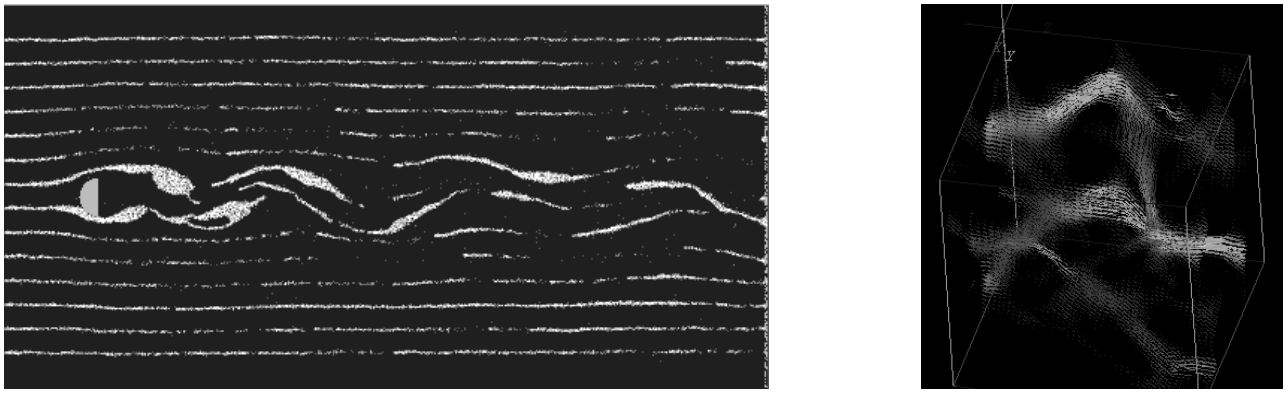

Figure 2: Discrete lattice molecular dynamics. (a) Hydrodynamic flow past a half-cylinder. Discrete particles flow in discrete directions at discrete speeds, on a spatial grid. We visualize the system using a second discrete fluid, a simulated tracer-gas (the visible "smoke") which "floats along" with the first fluid. This 2K×1K simulation runs at about 25 lattice-updates/sec on CAM-8. (b) Flow through a porous medium. The medium is a 3D bit-map derived by MRI from a real rock. This 256^3 simulation runs at about one complete (22-step) lattice-update per second on CAM-8.

vortex shedding from the obstacle. Figure 2b shows a lattice gas fluid flowing through porous sandstone. The boundary data for the simulation came from MRI of an actual piece of sandstone—the lattice gas interacts directly with the bit-mapped MRI data. Pressure and flow measurements using lattice gas algorithms have been compared with experimental data using the actual rock, and the results agree closely[24, 1].

Figure 3a shows measurements from a semi-numerical lattice gas simulation of an electromagnetic field scattering off a rectangular cylinder[6]. The graph compares a gas of 4-bit integers moving and interacting on a lattice with a standard finite difference calculation done on a lattice with four times as many lattice sites[8]. Figure 3b illustrates a phase change in a 3D dynamical Ising simulation of a magnetic material. Some of the bits at each lattice site in this simulation correspond to heat-bath variables. By controlling the average value of these variables, we control the temperature of the simulation[26, 19]. In a similar manner, CAM-8 has also been used extensively for 3D chemical reaction simulations[16], to study 3D crystallization of a gas into an elastic solid[28], and for the development and study of many other kinds of physical simulations[19].

Local lattice calculations have also been used extensively in image processing[25, 5]. With 1Tbit per second of memory/processing bandwidth available per embedded DRAM chip, ordinary video data rates are slow. Long and complicated SIMD programs could be run on each image in realtime. The physical simulation of Figure 3b was rendered as it ran by simulating digital light within the simulated material—much more elaborate rendering and volumetric image processing are made possible with faster processing. A technique closely related to rendering allows the lattice processors to compute only the wavefront of changing logic values in order to perform efficient large-scale logic simulations[18, 10]. Physically realistic "animation" of volumetic bit-map data, coupled with fast and sophisticated volume rendering, leads to many possibilities for complex 3D virtual-reality simulations, and agent based models.

3. DEALING WITH GRANULARITY

In this Section, we discuss how lattice computations such as those discussed above can be efficiently mapped onto the granular structure of memories. This issue will be addressed here in the context of writing a lattice algorithm for an ordinary computer, since the granularity constraints are similar to those that we must deal with using blocks of embedded DRAM.

By granular structure, we mean that the memory is organized into bytes, words and other bit-aggregates which are most efficiently accessed and used as units. For example, on an ordinary computer it is normally not possible to read just one bit of a memory word—at least one byte must be read. Consecutive bytes within a single memory word can typically be read more quickly than an equal number of bytes spread between several words. Words within a single cache-line can often be read more quickly than an equal number of words spread between several cache lines. Thus there may be a significant advantage in mapping our lattice data into memory in a manner that takes account of granularity constraints.

In Figure 4a, we present a very simple 1D example of a lattice computation. In this example, we have 16 lattice sites, with two bits of data at each site. The lattice sites are indicated by vertically oriented ovals, the data bits by black squares (top image). The *bit-field* consisting of the top bit from each lattice site is labeled A, while the bottom bit-field is labeled B. Now suppose we want to shift the A bit-field by two positions to the left, and the B bit-field by one position to the right. This is indicated in the middle image. In our example, the bits that shift past the ends of the lattice wrap around, and appear at the opposite ends. Once the data have all been moved, each lattice site is updated independently of all of the others, as is indicated in the bottom image. The two bits at each lattice site are used as inputs to some function, which produces new data for that lattice

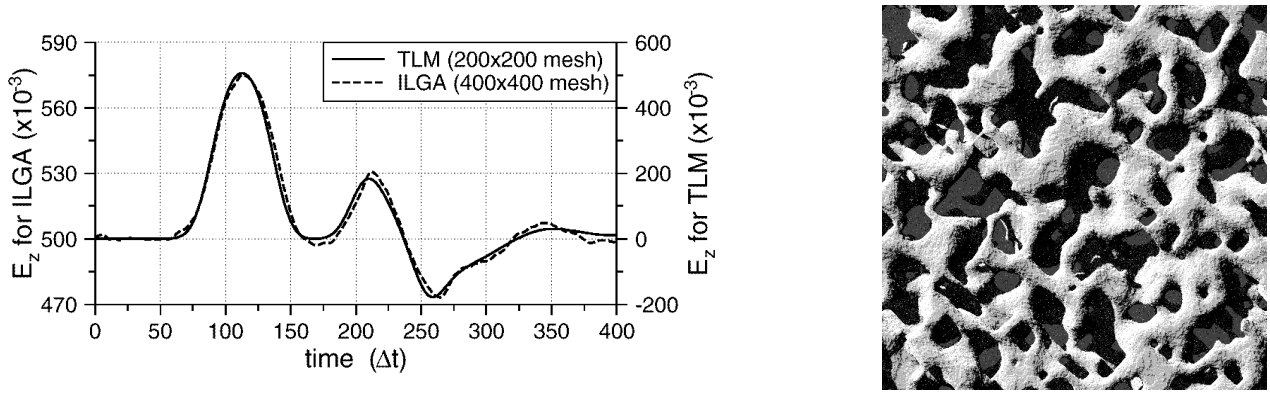

Figure 3: (a) Lattice-gas EM simulation. A simulation using a 4-bit Integer Lattice Gas Automaton (ILGA) of the TE_z scattered field from a PEC rectangular cylinder is compared with the standard TLM technique using floating-point variables. This 256^3 simulation also runs at about 1 lattice update per second. (b) Heat bath simulation of a phase change in a 3D magnetic substance. This 512×512×64 simulation runs at about 6 lattice-updates per second on CAM-8, including the volumetric processing to produce rendered images.

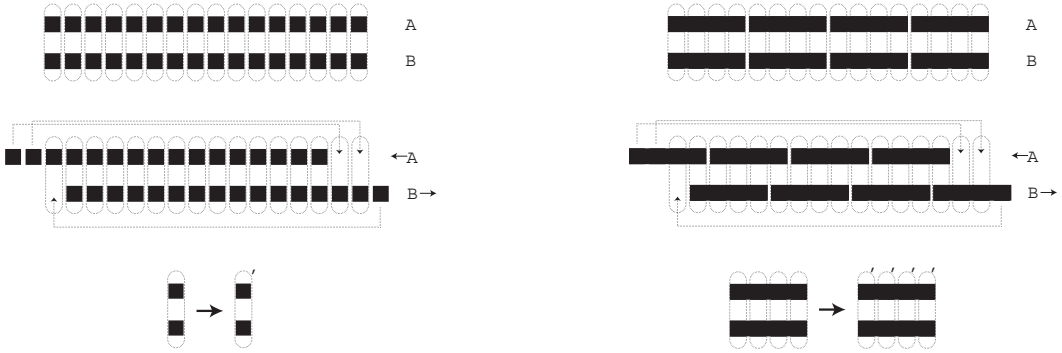

Figure 4: Data movement and processing example. (a) *Top:* 1D lattice with 2 bits at each of 16 lattice sites. All of the top bits form a bit-field (labeled *A*), and similarly with *B*. *Middle:* The *A* bit-field has been shifted two positions left, while the *B* bit-field has been shifted one position right. Data wraps around at the edges. *Bottom:* After all data has been moved, the data at each lattice site is independently processed. (b) Mapping the lattice data into memory words. *Top:* The lattice data is split up into 4-bit words. *Middle:* Shifting spoils the alignment of the edges of the 4-bit words, complicating assembling bit-field data for word-wide site-updating (*bottom*).

site—two new bits, in this example.

Now we would like to discuss implementing this algorithm as a program on an ordinary computer. To do this, we need to map the lattice data into the computer's memory. Let's suppose that the function we want to perform on each lattice site is some simple logic function that combines the pair of bits at that site. If pairs of bits are stored in pairs of DRAM words (at corresponding bit positions), then several sites can be processed in parallel by using word-wide logic operations. This suggests that we should put groups of A bit-field bits into one set of memory words, and put the corresponding groups of B bits into another set of memory words, with the bits in the same order. One way of doing this is shown in Figure 4b, top image. For illustrative purposes, we're assuming here that our memory word is only 4 bits long, and so in the Figure we've tied together groups of 4 bits. Unfortunately, when we shift the bit-fields (middle image), the groupings don't generally match, and so we have to do quite a bit of work to regroup bits before processing (bottom).

In Figure 5a, we indicate how we've divided the unshifted data in each bit-field into four memory words. In Figure 5b, we suggest a different grouping of bit-field bits into memory words that has nicer properties under shifts. Figure 6a shows the 16 lattice sites with the data unshifted, with each bit-field split up in this uniform manner into memory words. If we wanted to update the lattice data without shifting it, we would simply bring together the memory word $A[0]$ and the memory word $B[0]$ and perform word-wide logic on corresponding bits. Then we would operate on $A[1]$ and $B[1]$, etc. If, however, we want to update the shifted data, we can still process exactly the same four groups of lattice sites (i.e., the same *site-groups*) in exactly the same order (Figure 6b). The word $A[2]$ contains exactly the A bits that we need for processing the first site-group, and the word $B[3]$ has the B bits that we need. The rightmost $B[3]$ bit needs to wrap around, but this just means that we have to rotate the $B[3]$ word by one position before we use it. Similarly, all of the other site-groups can be processed by addressing the right words, rotating some of these words before using them.

Even though in this example we only process 2 bit-fields at a time, we can provide as many bits at each lattice site as we wish by simply allocating more bit-fields in memory. A succession of operations on pairs of bit-fields will implement any larger operation desired at every lattice site (though this is an inefficient way to perform complicated operations). Furthermore, when dealing with a single processor, it is easy to extend this 1D example to incorporate additional dimensions. The lattice can simply be split up into a set of 1D rows. Shifts along the rows are handled as described above; shifts in the additional dimensions only involve reordering the addresses of these 1D rows. Thus once again, all shifts *of any size and in any direction* can be accomplished by a combination of addressing plus word rotations.

Since our units of processing are groups of lattice sites that are spread evenly across the entire lattice, it is easy to perform multigrid and multiresolution computations. In a multigrid computation step, we only process a set of site-groups that comprise a power-of-two sublattice, and only use shifts that cause data belonging to those lattice sites to interact. In a multiresolution computation step, we cause two different power-of-two sublattices to interact. This is easily accomplished, since each higher resolution lattice is composed of several lower resolution lattices which turn into each other under shifts. Finally, we note that complicated crystalline lattices can be simulated using a collection of Cartesian lattices. As a simple example, the black squares on a checkerboard form a useful lattice. This lattice consists of two Cartesian sublattices: the black squares belonging to the even rows, and those belonging to the odd rows.

3.0.1 A hierarchy of constraints

Suppose that we have more than one level of granularity in our memory system. For example, suppose that our 4-bit words are most efficiently accessed as pairs of consecutive memory locations (doublewords). We can deal with this by adding a corresponding additional level of aggregation of bit-field bits. Referring to Figure 5b, we now group together the first and third sets of bits into a doubleword, still keeping each of the two 4-bit sets in separate words; and similarly for the second and fourth sets in another doubleword. Now all of the bits for each bit-field that belong to even-numbered lattice sites are in one doubleword, and all of the odd-site bits are in another. If we process all of the even lattice sites first, then depending on the shift amounts we will need one of these doublewords or the other for each bit-field. Within the even numbered lattice sites, we perform shifts as before using word-ordering and word rotations, and then similarly process the odd-numbered lattice sites.

In a similar manner, with a hierarchy of power-of-two lattice partitions we can match the bit-fields of a power-of-two sized lattice to a corresponding hierarchy of power-of-two granularity constraints. Since the data at each level is spread evenly across the entire lattice, shifts never regroup data at any level. Ultimately, all data movement is performed by a combination of memory addressing and rotation of the bits within the word addressed.

4. SIMD REVISITED

We have seen how to bring together sets of bits that all belong to the same site-group of lattice sites, to be processed simultaneously by word-wide operations. Since all lattice sites are processed identically, this is an ideal situation for SIMD processing. Since each bit-field needs to be addressed independently, bit-fields that are processed simultaneously should reside in separate blocks of memory. By processing many bit-fields at a time, we can make use of the full bandwidth of many blocks of memory. Alternatively, bit-fields can be accumulated sequentially, allowing them all to reside in a small number of blocks of memory. In this case, we can make use of many blocks of memory by putting multiple processing nodes on a single chip.

The addressing technique described above avoids any need to buffer words of memory data as they are accessed, and so it is particularly well suited to an embedded DRAM context, where bits of buffer memory may be enormously larger than DRAM cells. As we will see, we can also avoid most buffering associated with communications. By streaming lattice data past a set of SIMD processing elements, we can keep them busy at full data rate, and keep all memories operating at full

Figure 5: Two ways to map lattice data into memory. (a) The direct partition used in the previous Figure. Bit-field bits belonging to consecutive lattice sites are stored together in words of memory. (b) Bits spread evenly across the space are stored together in memory words. Each 4-bit memory word holds a uniform *skip-sample* of an entire bit-field.

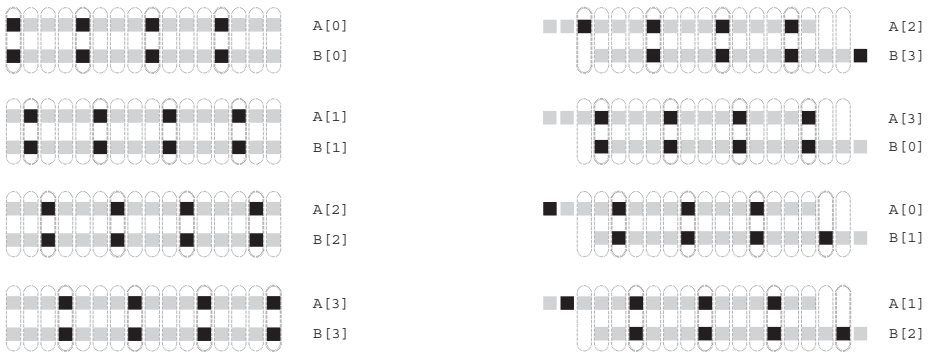

Figure 6: Data movement and processing using skip-samples. (a) For unshifted data, we could process 4 lattice sites at a time by bringing together pairs of memory words, and operating on pairs of corresponding lattice-site bits. (b) To bring shifted data together, we only need to change which memory words are accessed together. Notice that there is still some data that wraps around, but now in all cases wraparound happens within single memory words, and so each data word need only be rotated (if necessary) as it is read.

bandwidth. Thus we can use the full 1Tbit/sec bandwidth per chip that we discussed in the introduction. We will now describe how to connect together large arrays of such chips, all processing at full memory bandwidth.

4.1 Gluing sectors together

We will explain the systolic communication technique[15] used for interconnecting processors first in 1D. Then we will discuss processing in 2D and 3D before revisiting the topic of interprocessor communication.

As shown in Figure 1a, an emulated space is divided up evenly among a mesh array of processing nodes, with each node handling an equal sized *sector* of the space. In Figure 7, we show the sectors in three adjacent processors of a 1D mesh array. In a single-processor system, bit-field bits that spill past the edge of the sector (for example, **A** in the Figure) wrap around to the other edge, as shown. In a uniform bit-field shift across multiple processors, these same bits **A** should instead shift into the **B** position. Note, however, that the wrapped **A** and **B** bits are in the same position: a periodic shift within a sector puts all bits in the correct relative position within the sector; they are just in the wrong sector. Thus to construct a uniform shift across sector boundaries, we need only take bits that wrapped around within each sector, and substitute them for *the corresponding bits* in an adjacent sector. As long as all processors operate in lockstep, this means that any bit that wraps around should be communicated to the next processor, and substituted for the corresponding bit there, which that processor meanwhile sends to the processor next in line.

Thus it is wrapped bits that must be communicated, and there is never any delay (and associated buffering) needed before communication, since once all wrapped bits are substituted, the adjacent processor will have all of the bit-field data that it needs for the site-group that it is currently working on. The only issue that remains is balancing communications demand. In the 1D case, this has already been done, since each word of data is spread evenly across the entire sector handled by a given processor. For a given bit-field shift, the same fraction of each word will "stick out" past the end of the sector, and so need to be substituted for the corresponding bits that stick out in the adjacent sector.

4.2 2D and 3D

In 2D and 3D, balancing communications is not quite so automatic. If, for example, we simply constructed our 2D space out of a collection of horizontal 1D rows of data, then a given shift "down" would require communicating every bit belonging to a row at the bottom edge of the sector; whereas when processing a row that doesn't spill past the bottom edge, we would require no communication—thus the communication demand in the vertical direction would not be very even in time.

In Figure 8a, we show a way of partitioning a square 2D lattice (and hence all of the bit fields) so that communications demand is even in time. Recall that a site-group is a set of lattice sites that are processed together. In this example, we stretch one site-group evenly along the main diagonal of the sector. The other site-groups are then periodic shifts of this pattern. Clearly any periodic shift within the sector turns one site group into another. If a bit-field is partitioned into memory words that correspond to these site-groups, then shifted bit-field data needed for processing any site-group also all belongs to a single word of memory. In fact, if the bits within all memory words are ordered starting from the top of the sector, for example, then periodic horizontal shifts don't require any reordering of bits within words, while periodic vertical shifts can only rotate the bit order. Thus again all memory access just involves addressing plus rotating individual memory words as they are accessed.

The motivation for using a diagonal partition is that it treats horizontal and vertical shifts the same. For a given vertical or horizontal shift, the fraction of each memory word that shifts past the edge of the sector (and hence has to be communicated) is constant. Figure 8b shows the corresponding situation for a non-square sector, which is exactly analogous. Notice that for a pure horizontal shift, we can think of the picture in Figure 7 as being an edge-on view, and corresponding bits are simply substituted as usual. If we substitute horizontally before performing the vertical shift, then this again works exactly as before. Thus by pipelining the two substitution steps, we can shift in an arbitrary direction in 2D, using only nearest neighbor mesh connections. The situation in 3D is analogous, using site-groups that extend along the main diagonal of the cube and periodic shifts of this pattern. Gluing in 3D adds one level to our pipeline of bit substitutions. Bit-field shifting still only involves addressing plus rotation of individual words.

Note that adding dimensions makes little difference to our earlier discussion of how to deal with a memory granularity hierarchy. If we have a doubleword constraint, for example, we could group together the cases that are drawn above each other in Figure 8. This lattice partition remains *shift-invariant*: the constituent double site-groups turn into each other under shifts.

4.3 Hardware

The shifting and communication technique described above is suitable for a variety of spatial lattice architectures, with various kinds of SIMD processing elements and various numbers of simultaneously shifting bit-fields coupled to each PE. Here we describe hardware that is well suited for the kinds of non-numerical and semi-numerical simulations discussed in Section 2. We will also discuss in Section 4.4 how to modify this hardware, in order to adapt it to hybrid lattice computations that mix numerical and non-numerical data.

Figure 9a shows the major components and datapaths of a proposed chip, called SPACERAM, that uses the hardware resources illustrated in Figure 1b. About half of the chip area is taken up by 20 4Mb blocks of DRAM. A DRAM block, together with associated interface and mesh-I/O circuitry, constitutes a DRAM module. Each DRAM module has 64 I/O lines, all of which are used in the same direction at a given time to read or write a 64-bit data word. As is shown in Figure 9b, the n^{th} I/O line from each of the 20 modules is connected to the n^{th} PE. The PE's are all connected to a 2K-bit LUT-data bus. Each PE also has 2 bits of unbussed I/O associated with it. In addition to 48 differential-pair mesh interconnect signals, the chip has both master and slave RDRAM interfaces. This allows slave

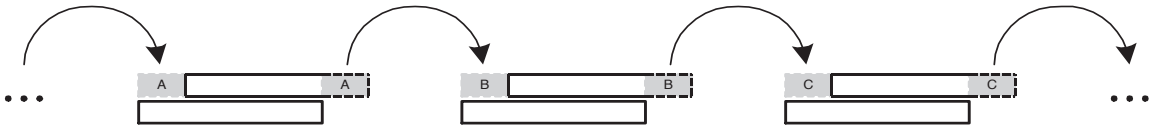

Figure 7: Gluing sectors together into a larger space.

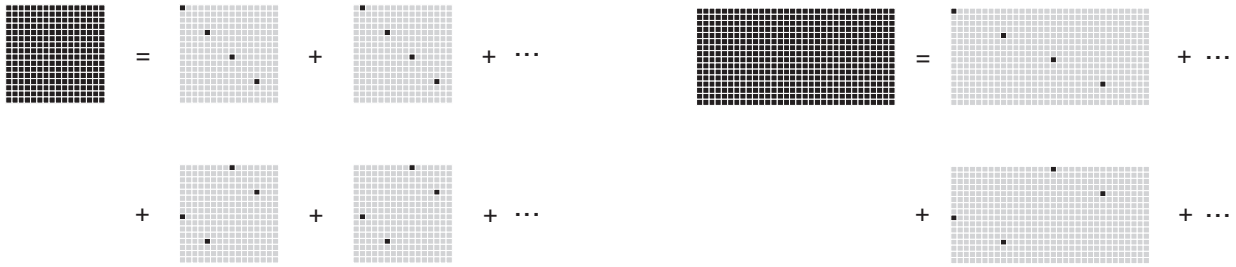

Figure 8: Partitioning a 2D sector into site groups.

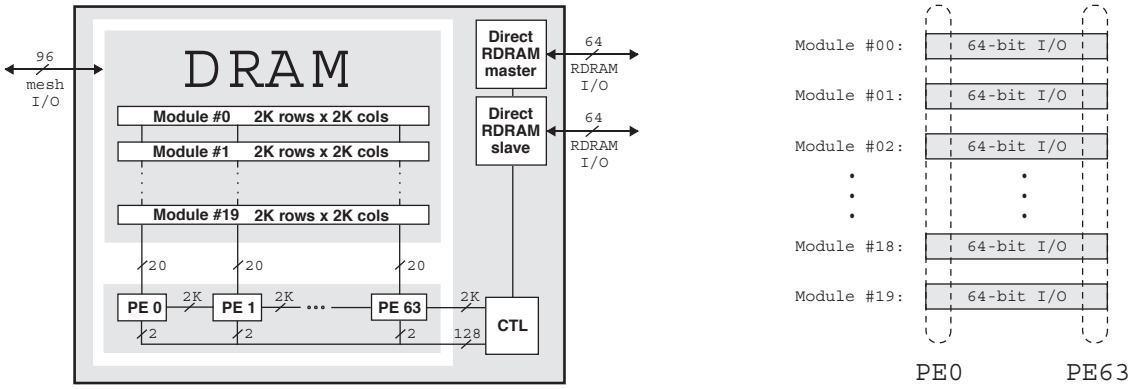

Figure 9: SPACERAM chip. (a) This is a schematic view of the resources shown in Figure 1b, indicating the main datapaths. Each of the 20 DRAM modules has 64 I/O's, one going to each of the 64 PE's. The PE's share 2K bits of bussed data, and each PE has 2 I/O lines that are controlled separately. Both master and slave RDRAM interfaces are provided, and 48 differential pairs are used for mesh interconnect. (b) PE's connect across DRAM module I/O words.

RDRAM's to be attached to each chip, in order to increase the size of simulations that can be run, and also allows the chip to be directly connected to a microprocessor that has an RDRAM master interface.

4.3.1 The DRAM module

Figure 10a shows the DRAM module in more detail. We are assuming here that the individual DRAM blocks are actually 2K×2K, with 256 data I/O lines running at 200MHz. The 64×4:64 MUX is used to narrow the datapath to 64 bits, and increase the clock rate to 800MHz. This reduces the area needed for wiring and multiplexes the 64 PE's across the wider raw data word, matching DRAM speed to logic speed. Since (for efficiency) all of each raw 256-bit data word must be used before moving on to another, this MUX simply adds an extra level to the memory granularity hierarchy of DRAM words and DRAM rows that our memory and communication organization is designed to deal with. Notice that if 800MHz is too aggressive, we simply change to a 128×2:128 MUX with twice as many PE's running at 400MHz. We can always adjust our virtualization ratio to match logic and DRAM speeds.

Each DRAM block can simultaneously access data from a different bit-field, each addressing periodically shifted data that all belong to the same 64-element site-group that is about to be processed by the 64 PE's. To complete the periodic shift within the sector, we may need to rotate the 64-bit word—this is done by the barrel rotator. Next we want to glue together the periodic shifts performed by the various mesh-connected SPACERAM chips. All bits that spill past the edge of the sector are put onto mesh-I/O wires, and are substituted for corresponding bits in an adjacent chip. The mesh I/O wires in each direction are a pooled resource that is shared by all DRAM modules. To shift a bit in the $+x$-direction, for example, the bit is put on an available $+x$ wire, and the bit coming in from the opposite direction on the corresponding $-x$ wire is substituted for it. Since all SPACERAM chips run in lockstep, they all make the same choice of which wire to use at a given time.

In this design, we provide the chip with 48 differential-pair mesh-I/O signals. This is enough to shift 8 bits at a time to or from each of 6 neighboring processors (3D mesh), 12 bits at a time with 4 neighbors (2D), or 24 bits at a time with 2 neighbors (1D). As is indicated in Figure 10a, the mesh-I/O unit puts a bit onto one of the 24 available output wires, and substitutes the corresponding bit from the corresponding input wire.

4.3.2 The processing element

Figure 10b shows the PE. As in our software example, the processing hardware is presented with aligned words of lattice data, with corresponding bits all belonging to the same lattice site. Thus each PE acts on data from a single site, and produces new bits that also belong to that site. In our software example, the processing involved only 2 simultaneously shifted bit-fields at a time, and was performed using word-wide logical operations. Here, we will instead operate on many more bit-fields at a time, and the processing will be based on lookup tables. Just as in our software example, however, all lattice sites in each site-group will be processed identically.

Figure 10b shows a single PE. The 20 I/O lines, one from each DRAM module, are attached to a permuting network. This network can attach any DRAM line to any of 20 different internal lines, so that any DRAM line can play any role. The permuters in all PE's are configured identically, so that all I/O bits of a given DRAM module are assigned to play the same role.

The basic element of each PE is an 8-input/8-output LUT. This size is based on our experience with symbolic calculations on CAM-8: this size speeds up our typical symbolic applications by about a factor of 100, compared to using a sequence of 2-input/1-output LUTs, as is common in simple SIMD machines.

Thus of the 20 DRAM I/O's, 16 are assigned as either a LUT input or output. Of the remaining 4 I/O's, 1 is used as a conditional bit: it determines whether the LUT is used, or the 8 bits belonging to that lattice site are instead left unchanged. This allows us to perform operations conditionally at different lattice sites, without using up half of our lookup table just for the case "do nothing." Of the remaining 3 bits, one is used for DRAM I/O (to or from external memory, for example), and two are used for control purposes.

4.3.3 The LUT

The box labeled "LUT" in Figure 10b is shown in more detail in Figure 10c. Since all PE's perform the same operation at the same time, they all need the same LUT data at the same time. An 8-in/8-out LUT has 2K bits of state, and so there are 2K bits of lut data that are bussed across all 64 PE's (as shown in Figure 9a). The actual lookup is performed by a 256×8:8 MUX that operates on this bussed data. The same data can also be used as an 11-in/1-out LUT, using some extra multiplexing (not shown).

By bussing the LUT data, we make it easy to change it quickly for all PE's simultaneously. Since (for efficiency's sake) all of a row must be processed once it is started, the row period is really an atomic unit of time in this architecture. We can allow the 2K bits of LUT data to be changed at the end of every row period (if desired) by devoting all of the bandwidth of just one DRAM module to this purpose during each row period. This involves the control signal labeled "next LUT data" in Figures 10b and 10c.

During each row period, 32 64-bit words are read or written from or to the 2K-bit row of each DRAM block. Thus each PE sees 32 bits per DRAM module. The bits coming in to the "next LUT data" input in each PE are stored in a 32-bit LUT data memory. This memory is double-buffered: while new LUT data is being loaded, each PE shows 32 bits of old LUT data on the 2K-bit *LUT bus*. When new LUT data is driven onto the LUT bus at the end of some 32-clock row period, processing waits for the data to settle before resuming.

Notice that it is possible to load bit-field data into the LUT instead of a precomputed table. This means that bits of spatial data can be randomly accessed by address, based on some of the bit-fields at each lattice site. This provides a powerful communications primitive for doing pointer following operations within each sector—something that would

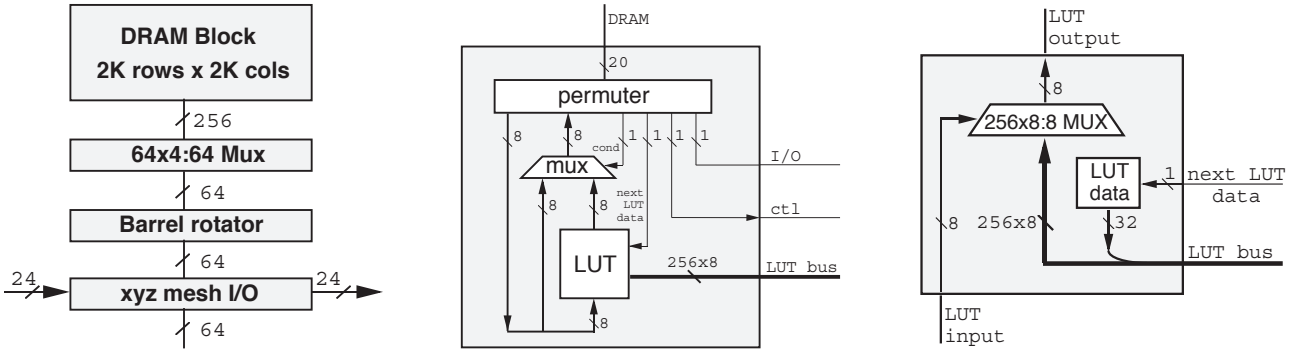

Figure 10: Major hardware components. (a) DRAM module. Together with each block of DRAM we group a MUX (which reduces the I/O width), a barrel rotator (to complete periodic shifts), and a mesh I/O interface (to substitute wrapped bits for corresponding bits in adjacent chips). (b) PE. This includes a permuter, which allows any DRAM module I/O bit to play any role internally, and an 8-in/8-out LUT, which transforms bits by table lookup. (c) The LUT is actually a MUX, indexing into data that is bussed across all 64 PE's. Each PE drives 32 bits of this bus.

otherwise be difficult in a SIMD architecture.

4.3.4 Control

During each row period, one DRAM row of one DRAM module provides 2K bits of control information that will be used during the next row period, including a specification of which DRAM module and row will provide the next 2K-bit control word. The control data includes a data direction, row address, starting column offset and rotation amount for each DRAM module, as well as a common setting for all permuters, mesh control data, and data for controlling the RDRAM I/O. All of this data is precomputed and loaded into the DRAM memory before processing begins. Linked lists of control words form a kind of microprogram, and a separate microprocessor can talk to all SPACERAM chips in order to specify which microprogram will be executed next. Some mesh I/O bandwidth can be stolen for this purpose, or the RDRAM slave interface can be used. A microprocessor can be included on each SPACERAM chip if more sophisticated microprogram control is desired.

4.3.5 External memory

As long as our SPACERAM chips are significantly more expensive, hold less data and consume more power than ordinary DRAM chips, there is a strong incentive to allow each SPACERAM chip to be augmented by a set of ordinary DRAM chips. This provision makes large 3D bit-mapped computations feasible. The external memory can also be used to hold extra program and LUT data, so that we can use very long and complicated programs, if necessary. If only program data is being obtained from external memory, this need not slow down the on-chip processing.

For very large lattice computations, we can treat on-chip memory as an additional level of memory hierarchy. By arranging to completely process a set of bit-fields for all lattice sites that are on-chip before moving on to another chip-scale site-group, we can often hide the enormous difference between on-chip and off-chip memory bandwidth. In general, if the processing at each lattice site is very complicated, as it often is in 3D physics simulations for example, then by completing a multi-step algorithm using a set of on-chip bit-field data corresponding to a chip-scale site-group, we may not waste any time at all waiting to swap data between the chip and external memory.

4.4 Numeric processing

An important application of lattice computation is numeric processing. It may also be very attractive to construct hybrid models where, for example, traditional numerical techniques are used to simulate the bulk of a material while a LUT-based lattice-gas molecular dynamics algorithm is used at the difficult-to-simulate interfaces.

Integer addition and subtraction can be performed efficiently using LUT's, but more complicated operations such as multiplication and division are rather slow. To multiply two k-bit integers using only the PE of Figure 10b, we need to pass each bit of each number in and out of DRAM about k times. With the addition of simple bit-serial arithmetic hardware, which includes data registers within the PE's, multiplication and division can be performed with only a single pass through DRAM, using the full memory bandwidth.

Bit-serial hardware receives its input bits sequentially. For example, to multiply two unsigned integers, we might first send the bits of the multiplicand into the serial multiplication unit one bit at a time. Then we would send the bits of the multiplier in one at a time, starting with the least significant bit (lsb). As the multiplier bits enter the multiplication unit, bits of the product leave the multiplication unit. The hardware inside the multiplication unit is very simple. It includes a register large enough to hold the multiplicand, an accumulator register of the same size that can shift by one position at a time, and an adder that can conditionally add the multiplicand into the accumulator, depending on the value of the current multiplier bit. When no additional multiplier bits remain, a new multiplicand can be loaded in while the final bits of the product are leaving. Division uses essentially the same hardware, and CORDIC algorithms for common transcendental functions are known which use similar hardware[21].

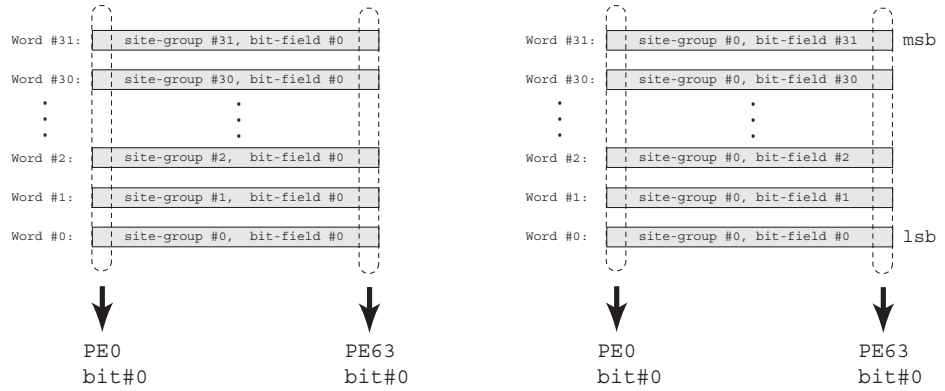

Figure 11: The structure of a DRAM row. (a) Format for symbolic calculations. (b) Format for numeric calculations.

In the processing hardware described above, we've seen that in one row-period each PE receives a sequence of 32 bits from each DRAM module. This is illustrated in Figure 11a, where we show an example of data coming into bit #0 of each PE. These 32 bits all belong to the same bit-field, and to 32 different lattice sites. In fact, each DRAM word belongs to a different 64-bit site-group of lattice sites.

If we could arrange for these bits to all come from the same lattice site, instead of from 32 different lattice sites, then our hardware would be perfectly organized for serial arithmetic. These 32-bits at each lattice site would constitute an integer stored there, and the PE need only incorporate a couple of registers and an adder/subtractor for each input bit-stream in order to efficiently perform multiplication and division. In Figure 11b, we show such a grouping of lattice data into a DRAM row. Here all words belong to the same 64-bit site group, each holding data from a different bit-field. This serial-arithmetic format is perfectly compatible with our bit-field shifting hardware of Figure 10a. Since all bit-fields within the row shift identically, we can perform shifts using row addressing in place of word addresssing. We complete the shift for each word within the row by rotating it if necessary, as usual. Since the grouping of bits into words hasn't changed, the rotation and substitution of bits within each word also hasn't changed. Using the LUT-based PE, data can be quickly converted between the two formats shown in Figure 11.

5. CONCLUSIONS

Spatially organized computation maps naturally onto spatial arrays of processing elements. For greatest density of information storage, it is natural today to use blocks of DRAM memory to hold the state of the computation. Granularity constraints in such memory do not hinder the efficiency of regular spatial algorithms. As our logic elements continue to become ever more microscopic, we expect that the attractiveness of massively parallel "crystalline" algorithms and architectures will only increase.

6. ACKNOWLEDGEMENTS

I thank Tom Knight for his support and encouragement, and for valuable suggestions. Jeremy Brown wrote a functional simulator for the SPACERAM chip. Amrit Pant studied the use of Benes networks for implementing the permuter in the PE. Greg Bridges and Neil Simons provided Figure 3a, and Dan Rothman provided Figure 2b.

7. REFERENCES

[1] C. Adler, B. Boghosian, E. Flekkoy, N. Margolus, and D. Rothman. Simulating three-dimensional hydrodynamics on a cellular-automata machine. *Journal of Statistical Physics*, 81:105–128, Oct 1995.

[2] B. Boghosian, F. Alexander, and P. Coveney, editors. *Discrete Models of Complex Fluid Dynamics*, published as a special issue of the International Journal of Modern Physics C, volume 8(4), 1997.

[3] B. Boghosian, P. Coveney, and A. Emerton. A lattice-gas model of microemulsions. *Proceedings of the Royal Society of London A*, 452:1221–1250, 1996.

[4] B. Boghosian, J. Yepez, F. Alexander, and N. Margolus. Integer lattice gases. *Physics Review E*, 55(4):4137–4147, 1996.

[5] M. Bolotski, R. Amirtharajah, W. Chen, T. Kutscha, T. Simon, and T. Knight, Jr. Abacus: A high performance architecture for vision. In *International Conference on Pattern Recognition*, Oct 1994.

[6] G. Bridges and N. Simons. Extremely low-precision integer cellular array algorithm for computational electromagnetics. *IEEE Microwave and Guided Wave Letters*, 9(1):1–3, Jan 1999.

[7] S. Chen, G. Doolen, K. Eggert, D. Grunau, and E. Loh. Lattice gas simulations of one and two-phase fluid flows using the Connection Machine-2. In A. Aves, editor, *Discrete Models of Fluid Dynamics, Series on Advances in Mathematics for Applied Sciences*, volume 2, page 232. World Scientific, 1991.

[8] C. Christopoulos. *The Transmission Line Modelling Method (TLM)*. IEEE Press, 1993.

[9] P. Christy. Virtual processors considered harmful. In Q. Stout and M. Wolfe, editors, *The Sixth Distributed Memory Computing Conference Proceedings*, pages 99–103. IEEE Computer Society Press, 1991.

[10] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, 1996. Reprinted as MIT AI Lab Technical Report 1586.

[11] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Physics Review Letters*, 56:1505–1508, 1986.

[12] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The terasys massively parallel PIM array. *IEEE Computer*, pages 24–31, Apr 1995.

[13] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[14] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, et al. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[15] H. T. Kung. Systolic communication. In *Proceedings of the International Conference on Systolic Arrays*, May 1988.

[16] A. Malevanets and R. Kapral. Microscopic model for Fitzhugh-Nagumo dynamics. *Physics Review E*, 55(5):5657–5670, 1997.

[17] N. Margolus. CAM-8: A computer architecture based on cellular automata. In A. Lawniczak and R. Kapral, editors, *Pattern Formation and Lattice-Gas Automata*, pages 167–187. American Mathematical Society, 1996.

[18] N. Margolus. An FPGA architecture for DRAM-based systolic computations. In Arnold et al., editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 2–11. IEEE Computer Society Press, 1997.

[19] N. Margolus. Crystalline computation. In A. Hey, editor, *Feynman and Computation*, chapter 18. Perseus Books, 1999.

[20] N. Margolus, T. Toffoli, and G. Vichniac. Cellular-automata supercomputers for fluid dynamics modeling. *Physics Review Letters*, 56:1694–1696, 1986.

[21] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, 1997.

[22] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *The 25th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 1998.

[23] S. Reddaway. Signal processing on a processor array. In Lacoume et al., editors, *Traitement Du Signal / Signal Processing*, volume 2. Elsevier Science, 1987.

[24] D. Rothman and S. Zaleski. *Lattice-Gas Cellular Automata—simple models of complex hydrodynamics*. Cambridge University Press, 1997.

[25] S. Tanimoto and J. Pfeiffer, Jr. An image processor based on an array of pipelines. In *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, volume 81, pages 201–208, 1981.

[26] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, 1987.

[27] H. Wijshoff and J. van Leeuwen. The structure of periodic storage schemes for parallel memories. *IEEE Transactions on Computers*, c-34(6):501–505, 1985.

[28] J. Yepez. Lattice-gas crystallization. *Journal of Statistical Physics*, 81:255–294, 1994.