Complex Systems **1** (1987) 939–965

Performance of VLSI Engines for Lattice Computations*

Steven D. Kugelmass
Richard Squier
Kenneth Steiglitz
Department of Computer Science, Princeton University, Princeton, NJ 08544, USA

Abstract. We address the problem of designing and building efficient custom VLSI-based processors to do computations on large multi-dimensional lattices. The design tradeoffs for two architectures which provide practical engines for lattice updates are derived and analyzed. We find that I/O constitutes the principal bottleneck of processors designed for lattice computations, and we derive upper bounds on throughput for lattice updates based on Hong and Kung's graph-pebbling argument that models I/O. In particular, we show that $R = O(BS^{1/d})$, where R is the site update rate, B is the main memory bandwidth, S is the processor storage, and d is the dimension of the lattice.

1. Introduction

This paper deals with the problems of designing and building practical, custom VLSI-based computers for lattice calculations. These computational problems are characterized by being iterative, defined on a regular lattice of points, uniform in space and time, local, and relatively simple at each lattice point. Examples include numerical solution of differential equations, iterative image processing, and cellular automata. The recently studied lattice gas automata, which are microscopic models for fluid dynamics, are proposed as a test bed for the work.

The machines envisaged—lattice engines—would typically consist of many instances of a custom chip and a general-purpose host machine for support. In many practical situations, the performance of such machines is limited,

*This work was supported in part by NSF Grant ECS-8414674, U.S. Army Research Office–Durham Contract DAAG29-85-K-0191, and DARPA Contract N00014-82-K-0549. An earlier version of this paper appears in [7].

not by the speed and size of the actual processing elements, but by the communication bandwidth on- and off-chip and by the memory capacity of the chip.

A familiar example of lattice-based computational tasks is two-dimensional image processing. Many useful algorithms, such as linear filtering and median filtering, recompute values the same way everywhere on the image, and so are perfectly uniform; they are local in that the computation at a given point depends only on the immediate neighbors of the point in the two-dimensional image.

Another class of calculations, besides being uniform and local, has the additional important characteristic of using only a few bits to store the values at lattice points, and so is extremely simple. Further, the calculations operate on local data iteratively, which means that they are not as demanding of external data as many signal processing problems. These computational models—uniform, local, simple, and iterative—are called *cellular automata*. We will next describe a particular class of cellular automata, one that provides a good test bed for the general problems arising in the design of dedicated hardware for lattice-based computations.

2. A paradigm for lattice computations: the lattice gas model

Quite recently, there has been much attention given to a particularly promising kind of cellular automaton, the so-called *lattice gases*, because they can model fluid dynamics [14]. These are lattices governed by the following rules:

> At each lattice site, each edge of the lattice incident to that site may have exactly zero or one particle traveling at unit speed away from that site, and, in some models, possibly a particle at rest at the lattice site.
>
> There is a set of collision rules which determines, at each lattice site and at each time step, what the next particle configuration will be on its incident edges.
>
> The collision rules satisfy certain physically plausible laws, especially particle-number (mass) conservation and momentum conservation.

These lattice gas models have an intrinsic exclusion principle, because no more than one particle can occupy a given directed lattice edge at any given time. It is therefore surprising that they can model fluid mechanics. In fact, in a two-dimensional hexagonally connected lattice, it has been shown that the Navier-Stokes equation is satisfied in the limit of large lattice size. This model is called the FHP model, after Frisch, Hasslacher, and Pomeau [3]. The older HPP model [4], which uses an orthogonal lattice, does not lead to isotropic solutions.

The idea of using hexagonal lattice gas models to predict features of fluid flow seems to be about two years old, and whether the general approach of simulating a lattice gas can ever be competitive with more familiar numerical solution of the Navier-Stokes equation is certainly a premature question.

Extensions to three-dimensional gases are just now being formulated [1], and quantitative experimental verification of the two-dimensional results is fragmentary. The Reynolds Numbers achievable depends on the size of the lattices used, and very large Reynolds Numbers will require huge lattices and correspondingly huge computation rates. For a discussion of the scaling of the lattice computations with Reynolds Number, see [10].

What is clear is that the ultimate practicality of the approach will depend on the technology of special-purpose hardware implementations for the models involved. Furthermore, the *uniformity*, *locality*, and *simplicity* of the model mean that this is an ideal test bed for dedicated hardware that is based on custom chips. We will therefore use the lattice gas problem as a running example in what follows. We especially want to study the interaction between the design of custom VLSI chips and the design of the overall system architecture for this class of problems.

We will present and compare two competing architectures for lattice gas cellular automata (LGCA) computations that are each based on VLSI custom processors. The analysis will focus on the permissible design space given the usual chip constraints of area and pin-out and on the achievable performance within the design space. Following this, we will present some theoretical upper bounds for the computation rate over a lattice, based on a graph-pebbling argument.

3. Serial pipelined architectures for lattice processing

We are primarily interested in special-purpose, VLSI-based processor architectures that have more than one PE (*processing element*) per custom chip. It is important to recognize that if the PEs are not kept busy, then it might be more effective (in terms of overall throughput) to have fewer PEs per chip but to use them more efficiently. Although there are many architectures that have the property of using PEs efficiently, we will only describe two, both based on the idea of serial pipelining (see figure 1). This approach has the benefit that the bandwidth to the processor system is small even though the number of active PEs is large. This serial technique has been used for image processing where the size of the two-dimensional grid is small and fixed [6,13,17] and has also been used to design a high-performance custom processor for a one-dimensional cellular automaton [16].

Consider what is required to pipeline a computation. We must guarantee that the appropriate site values of the correct ages are presented to the computing elements. In the case of the LGCA, we can express this data dependency as:

$$v(a, t+1) = f(N(a), t)$$

where $v(a,t)$ is the value at lattice site a at time t, $N(a)$ is the *neighborhood* of the lattice site a, and f is the function that determines the new value of a based on its neighborhood. The LGCA requires all the points in the neighborhood of a to be the same age in order to compute the new value,

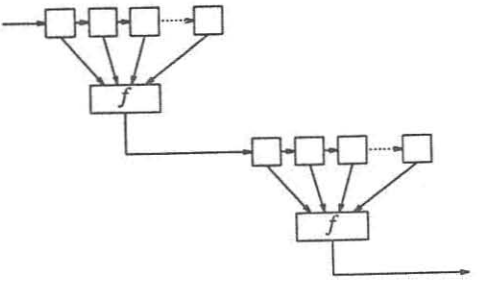


Figure 1: One-dimensional pipeline.

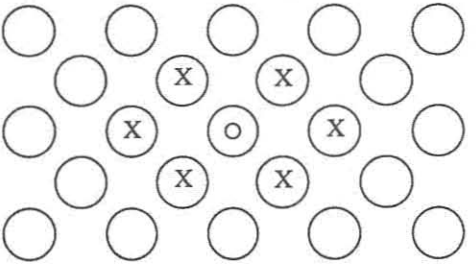


Figure 2: Hexagonal neighborhood. The circled site is a; the sites with Xs constitute its neighborhood.

$v(a, t+1)$. The LGCA has a neighborhood that looks like the example given in figure 2.

One-dimensional pipelining also requires a linear ordering of the sites in the array. That is, we wish to send the values associated with the sites one at a time into the one-dimensional pipeline and receive the sequence of sites in the same order possibly some generations later. Therefore, we would like sites that are close together in the lattice to be close together in the stream. In this way, the serial PE requires a small local memory because neighborhoods (sites that are close together in the array) will also be close together in the stream. Unfortunately, the lattice gas automaton can require a large amount of local memory per PE because there is no sublinear embedding of an array into a list [12].

The natural row-major embedding of the array into a list preserves 2-

neighborhoods[1] with diameter $2n-2$. This means that a full neighborhood of a site from an $n \times n$ lattice is distributed in the list so that some elements of the neighborhood are at least $2n-2$ positions apart. This embedding is undesirable for two reasons. The amount of local memory required by a PE is a function of the problem instance, forcing us to decide in advance the size of one dimension of the lattice (one can actually process a prism array, finite in all but one dimension) because the chip will only work for a single problem size due to its fixed span. The second deficiency is due to the size of the span. If $n = 1000$, then each PE would require about 2000 sites worth of memory. This puts a severe restriction on the number of PEs that can be placed on a chip.

Unfortunately, the $2n-2$ embedding is optimal. Rosenberg showed this bound holds for prism array realizations but it has been unknown whether it is possible do better for finite array realizations. Rosenberg's best lower bound for the finite array case has never been achieved and he suspected that the row-major scheme was optimal. Sternberg [18] also questioned whether or not the storage requirement for a serial pipelined machine could be reduced. Supowit and Young [19] showed that the row-major embedding is optimal and therefore a serial pipeline must use at least $2n-2$ storage.

Theorem 1. *Place the numbers* $1, \ldots, n^2$ *in a square array* $a(i,j)$*, and define the span of the array to be*

$$\max\{|a(i+1,j) - a(i,j)|, |a(i,j+1) - a(i,j)|\}$$

Then span $\geq n$.

Proof. Put the numbers in the array one at a time in order, starting with 1. When for the first time there is either a number in every row or a number in every column, stop. Without loss of generality, assume this happens with a number in every row.

We claim that there cannot be a full row. Suppose the contrary. The last number entered was placed in an empty row, so there must have been a full row before we stopped. This would mean there was a number in every column before there was a number in every row.

Since there is no full row, but a number in every row, there is at least one vacant place in every row that is adjacent to an occupied spot. Choose one such vacant place in each row, and call them set F (with $|F| = n$). Now, if we stopped after placing the number t, the places in F will get filled with numbers greater than t. The largest number that will be put in a location in F is $\geq t+n$, and will be adjacent to a number $\leq t$. ∎

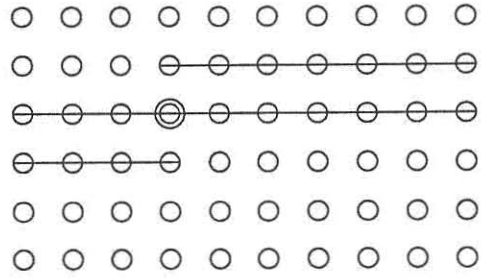
The critical system parameters for the one-dimensional pipeline architecture, system area and total system throughput, can be varied over a range of values. The actual selection of the operating point on the throughput-area curve depends on several factors: for example, the problem instance size and total system cost.

[1] Sites that are two edge traversals apart in the lattice.

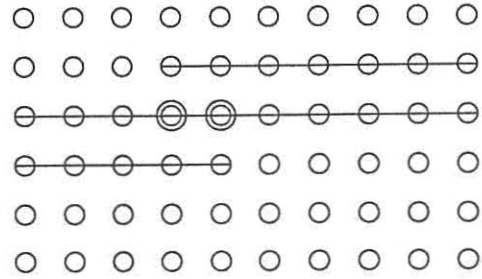
The appealing aspects of the serial architecture are the simplicity of its design, its small area in comparison to other architectures, and the small input/output bandwidth requirement. The computation proceeds on a wavefront [8] through time and space, each succeeding PE using the data from the previous PE without the need for further external data.

4. Wide-serial architecture (WSA)

Throughput in a serial architecture can be improved by adding concurrency at each level of the pipeline. One way to accomplish this is to have each pipeline stage compute the new value of more than one site each clock period. For example, if the computation at PE j is at the point where site a, circled, is to be updated, then PE j contains the data indicated by strike-out in the following:



We could allow a second PE j' to compute site $a+1$ at the same time if we store just one more data point.



The most attractive feature of this scheme is that performance is increased, but at a cost of only the incremental amount of memory needed to store the extra sites. The on-chip memory per PE is also improved dramatically; it decreases linearly with the number of PEs per chip. However, there is a price to pay: two new site values are required every clock period so that two site updates can be performed. The extra PEs require added bandwidth to and from the chip and this implies that the main memory system must provide that bandwidth as pins or wires.

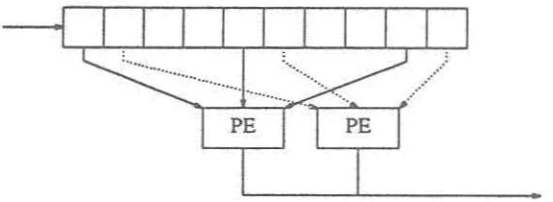


Figure 3: Wide-serial architecture.

As an example, the following figure shows how two PEs on the same chip can cooperate on a computation. Each square of the shift register holds the value of one site in the lattice. Every clock period, two new site values are input to the chip, two sites are updated, and their values are output to the next chip in the pipeline.

5. Sternberg partitioned architecture (SPA)

In reference [18], Sternberg proposes that a large array computation can be divided among several serial processors, each of which operates as described earlier. The array is divided into adjacent, non-overlapping columnar slices, and a fully serial processor is assigned to each slice (see figure 4).

The processors are not exactly the same as those described above; they are augmented to provide a bidirectional synchronous communication channel between adjacent partitions so that sites whose neighborhoods do not lie entirely in the storage of a single PE can be computed correctly and in step with other site updates. See reference [18] for details.

Dividing the work in this way accomplishes three things. First, it decreases the amount of storage that each PE needs in order to delay site values for correct operation of the pipeline. This comes about because each PE needs to delay only two lines of its slice, not the whole line width. Second, it increases the ratio of processing elements to the total number of sites, permitting an increase in the maximum throughput by a multiplicative constant equal to the number of slices. Third, it provides a degree of modularity and extensibility. It is possible to join two smaller machines along an edge to form a machine that handles a larger problem.

In the case of a VLSI implementation, decreasing the size of the local storage is extremely important because most of the silicon area in the implementation of a serial processor is shift register. Since each PE in the SPA architecture requires fewer shift register storage cells, it is possible to place several PEs on a chip, whereas if each serial PE were required to store two lines of the whole lattice, then only one or two PEs could be placed on a

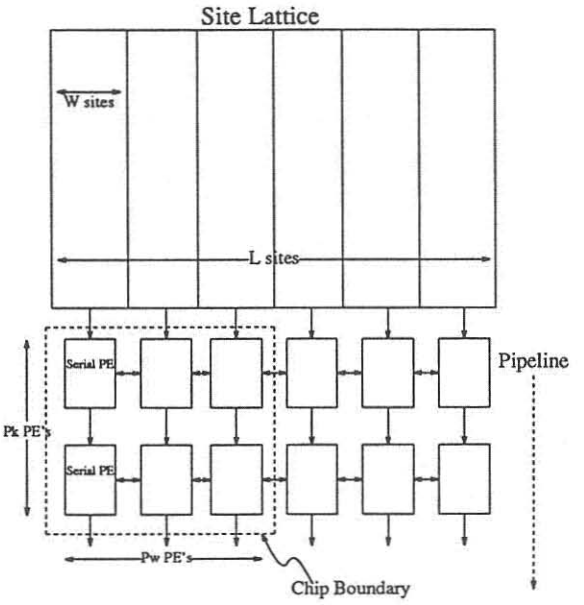


Figure 4: Sternberg partitioned architecture.

VLSI chip with current technology. The only way around this limitation is to use another technology to implement the required storage, such as off-chip commercial memories, in which case we quickly encounter pin limitations.

It is important to recognize that the *total* amount of storage required under this organization is two lines of the whole lattice per pipeline stage. Thus, the total storage requirement under this implementation is not reduced below that of the fully serial approach presented earlier. We should also not forget that each column of serial processors requires its own data path to and from main memory. This data path is a relatively expensive commodity. In fact, as we will see in the upcoming analysis, the data path is the most expensive commodity in a VLSI implementation of this architecture.

The analysis will demonstrate an underlying principle of VLSI implementations of architectures for multi-dimensional spatial updates, namely that I/O pins are the critical resource of a VLSI chip.

6. Analysis and comparison of WSA and SPA

In this section, we analyze and compare the Sternberg partitioned architecture (SPA) with the wide-serial architecture (WSA) that we proposed in section 4. The analysis derives the optimum throughput and area of processing systems composed of VLSI chips for the two-dimensional FHP lattice gas problem. We define the design parameters for each system and derive the design curves and optimum values of those parameters. For the analysis, we

assume that a memory system capable of providing full bandwidth to the processor system is available.[2] Finally, we compare the systems on the basis of maximum throughput, total system area, and throughput-to-area ratio. We also discuss the relative advantages and disadvantages of both architectures with an emphasis on system complexity and ease of implementation.

6.1 Wide-serial architecture (WSA)

The WSA has system parameters: (assumes 1 pipeline stage per chip, P processing elements wide)

$$N = k \text{ chips} \qquad \text{(System Area)}$$
$$R = F \cdot P \cdot k \frac{\text{sites}}{\text{second}} \qquad \text{(System Throughput)}$$

and chip constraints

$$2D \cdot P \leq \Pi \qquad \text{(Chip Pins)}$$
$$\beta(2L + 7P + 3) + \gamma P \leq \alpha \qquad \text{(Chip Area)}$$

where

N is the total number of chips constituting the processor,

P is the number of PEs per VLSI chip,

k is the total depth in PEs of the processor pipeline (path length),

F is the major cycle time of the chip,

D is the number of bits required to represent the state of a lattice site,

L is the number of sites along an edge of the square lattice,

PI is the total number of pins usable for input/output,

β is the area of a shift register that holds a site value, in λ^2,

γ is the area of a PE, in λ^2,

α is the total usable chip area, in λ^2.

For convenience, we also define:

$$B = \frac{\beta}{\alpha} = \text{ normalized site storage area}$$
$$\Gamma = \frac{\gamma}{\alpha} = \text{ normalized processor area}$$

Less formally, this says that the number of chips that we need for the processor equals the total pipeline depth required, k. The processing rate that this system achieves is equal to the depth of the pipeline, multiplied by the number of processors at each depth, multiplied by the rate at which a processor computes new sites. We are assuming that each VLSI chip will

[2]This is a very important assumption.

contain only a single wide parallel pipeline stage. That is, the chip is not internally pipelined with wide-serial processors.

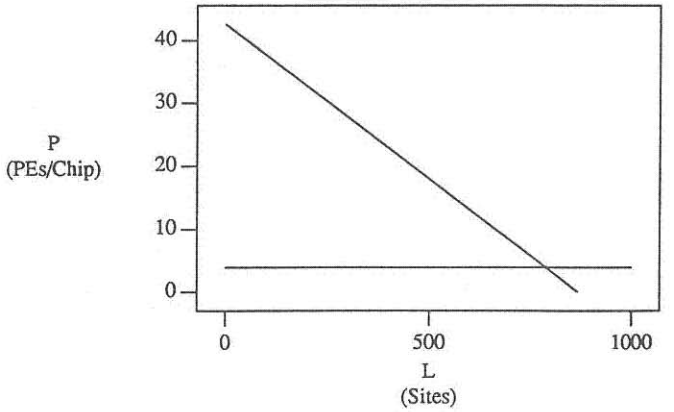
We wish to maximize R subject to having a fixed number of chips, $N = N_0$, and subject to constraints on the pin count and area of the VLSI custom chip. Notice that the problem is equivalent to maximizing P subject to the chip constraints because $R = F \cdot P \cdot k = F \cdot P \cdot N$, where F and N are fixed (N is fixed at N_0).

The constraints are described in the $L - P$ plane by the following two inequalities:

$$P \leq \frac{\Pi}{2D}$$

$$P \leq \frac{1 - 3B - 2BL}{7B + \Gamma}$$

If we consider an example where $D = 8$, $\Pi = 72$, $B = 576 \times 10^{-6}$, and $\Gamma = 19.4 \times 10^{-3}$ (figures derived from our actual layouts) we get the following graph:



The chip constraints require that the operating point determined by P and L lie below both curves. The intersection of the two curves is $P \approx 4$ and $L \approx 785$. Beyond that point, we need to decrease the number of processors on a chip to make room for more memory—an undesirable situation because throughput then drops off linearly. Furthermore, we want L to be as big as possible, so the corner is the logical choice of operating point.

We are also interested in the ultimate maximum performance that the architecture can deliver using any number of chips. It is easy to see that the maximum throughput for a fixed clock frequency, F, comes when the pipeline depth, k, is at a maximum. A maximum value, $k_{\max} = L$, arises because at that point the pipeline contains all the values of the sites in the lattice and there is no new data to introduce into the processor pipeline. The

maximum values for processor system area and processor system throughput are therefore:

$$N_{\max} = L \text{ chips}$$

$$R_{\max} = \frac{\Pi}{2D} \cdot F \cdot L \frac{\text{sites}}{\text{sec}}$$

It is also interesting to note that there is an upper bound on L even if we were to accept arbitrarily slow computation. At a certain point all the chip area would be used for memory, leaving no room for PEs.

The major limitation of this architecture is that the largest problem instance is fixed by the chip technology, but it has the redeeming features of simplicity, ease of implementation, and small main memory bandwidth.

6.2 Sternberg partitioned architecture (SPA)

This processor computes updates for a lattice L sites on a side by partitioning the lattice into non-overlapping slices that are each W sites wide (there are $\frac{L}{W}$ such slices). Each of the VLSI chips that compose the processor computes P_w slices and the computation of each slice is pipelined on the chip to a depth P_k (see figure 4). It is then easy to see that the system has area and throughput:

$$N = \frac{L/W}{P_w} \cdot \frac{k}{P_k} \text{ chips} \qquad \text{(System Area)}$$
$$R = F \cdot k \cdot \frac{L}{W} \frac{\text{sites}}{\text{sec}} \qquad \text{(System Throughput)}$$

To derive the constraints on the VLSI chip, notice that the communication path between chips in the direction of the data pipeline requires $2DP_w$ pins, and that the "slice-to-slice" path requires $2EP_k$, where E is the number of bits required to complete the information contained in a single site's neighborhood when that neighborhood is split across a slice boundary. However, the chip must use no more than α area, of which processors each require γ, and memory to hold a site value requires β. Thus, the whole chip is governed by the constraints

$$2DP_w + 2EP_k \leq \Pi \qquad \text{(Chip Pins)}$$
$$((2W+9)\beta + \gamma)P_wP_k \leq \alpha \qquad \text{(Chip Area)}$$

We again wish to maximize throughput with respect to a fixed number of chips, $N = N_0$, while at the same time satisfying the VLSI chip constraints of area and bandwidth. This again turns out to be equivalent to maximizing the total number of processors on the chip because we can easily verify by direct substitution that $R = F \cdot k \cdot \frac{L}{W} = P_wP_k \cdot F \cdot N_0$. Since F and N_0 are fixed, it suffices to maximize the product $P_wP_k = P$ subject to the constraints above.

To evaluate the design space of SPA, it is helpful to view it in the $W - P$ plane. We do this via a change of variables:

$$P = P_wP_k.$$

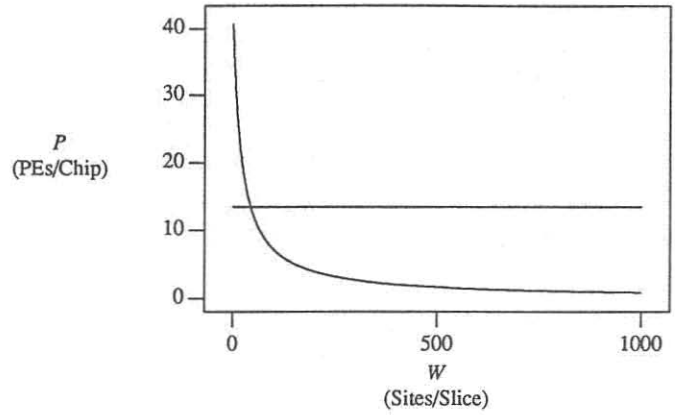
Rewriting the chip inequalities yields

$$2DP_w + 2E\frac{P}{P_w} \leq \Pi$$

$$((2W+9)B+\Gamma)P \leq 1$$

where P_w, P, and W are variables. This is the logical choice of variables for this architecture because they are the ones that are constrained by the chip technology and govern the optimal design of the chip. Once we know good values for them, a machine which can compute for an arbitrary lattice width L can be built by increasing the number of slices of width W.

When these curves are projected onto the $W - P$ plane using the values for D, Π, B from the previous example, and setting E to 3 (three bits must be passed to complete a neighborhood), we have



The constant curve is a projection of the first constraint where P_w is given the value which permits P to achieve its maximum value. For this example, this occurs at $P_w = \frac{9}{4}$. As before, we need to operate below both curves, and the corner at $P \approx 13.5$ and $W \approx 43$ yields the best choice. Beyond this point, throughput drops off quite rapidly as the silicon real estate is used by memory.

6.3 Discussion

The above analysis gives us two different viewpoints from which to make a comparison between these two architectures. Ignoring extensibility, we can make a comparison between the two designs when they are optimized for throughput, as they were in the preceding. Taking a more general point of view, we can make the comparison by using a slight variant of WSA which allows for extensiblity by sacrificing processing speed.

First, let us compare the designs optimized for throughput without regard to extensibility. The optimal WSA configuration limits the lattice length

to $L = 785$. Both WSA and SPA systems have throughput rates which grow linearly with the number of chips. However, SPA is three times faster than WSA. (SPA has twelve processors per chip while WSA has four.) On the other hand, the SPA system requires four times as much main memory bandwidth as the WSA system: 262 bits/tick versus 64 bits/tick.

The above argument contains a bias in favor of SPA. System timing is an important consideration which can make it difficult to clock SPA as fast as WSA. The WSA architecture has connectivity only in one dimension, whereas the SPA system requires communication in both the pipeline direction and the synchronous side-to-side data paths. This added complexity is a more pronounced drawback for SPA when extensibility is considered, as we will mention below. The conclusion in both cases favors the WSA system when it comes to considering an implementation. There is also the matter of the data access pattern in the memory. The WSA machine accesses the data in a strict raster scan pattern which is simpler than the row-staggered pattern that the SPA scheme requires for its operation.

The SPA architecture has one considerable advantage over the WSA scheme: extensibility. Smaller instances of an SPA machine can be joined together to form a machine that computes a larger lattice. This is not true for the WSA case, where computation is limited to lattice sizes which do not exceed L as given by the chip area constraint, because all the required data must fit on the chip. This requirement is relaxed in the SPA scheme because data can be moved between adjacent chips as W is adjusted to the chip constraints and an arbitrary lattice width L can be supported by composing a suitable number of slices. In this respect, the two schemes seem incomparable.

Our second point of view on the comparison of these two architectures is facilitated by considering a slight variation of WSA which allows extension of the lattice size. The extension can be accomplished by moving a portion of the shift register off chip. The pin constraints given previously, with the same constants, allow only one processor per chip in this case. A stage in the pipeline consists of a processor chip and associated shift registers sufficient to hold the remainder of the $2L + 10$ node values which do not fit onto the processor chip. We will call this version of WSA WSA-E.

Both SPA and WSA-E systems have throughput rates that grow linearly with the number of chips in the system for a fixed lattice size L. However, the constant of proportionality between the two rates grows with increasing lattice size. The reason is that the number of processors per unit chip area is independent of lattice size for SPA, whereas it decreases with increasing lattice size for WSA-E. So, for instance, given the same number of chips and a lattice size $L \leq 785$, the SPA system is twelve times faster than WSA-E because it has twelve processors per chip as opposed to one per chip.

A better understanding of the contrasts between the two systems can be obtained by looking at requirements for main memory bandwidth and storage area per processor. WSA-E has a constant bandwidth requirement of 16 bits per clock tick and requires $(2L + 10)B$ storage area per processor; SPA has a

main memory bandwidth requirement of $\frac{L}{3}$ bits per tick and requires $(1284)B$ area per processor. For a fixed processing rate, the penalty for larger lattice size is either linear growth in the number of chips for the WSA-E system, or linear growth in the main memory bandwidth in the SPA case. For example, if $L = 1000$, then WSA-E requires about twice as much area as SPA, while requiring about one twentieth as much bandwidth.

6.4 Summary

We have analyzed the critical parameters of two system architectures for high performance computation on a cellular automaton lattice. We see that the WSA architecture offers good throughput at a modest system area and complexity, while the SPA architecture offers higher performance, at the price of increased complexity and memory bandwidth.

The preceding analysis suggests that the ultimate limit to the performance of these architectures, and any alternatives, will stem from chip pin-bandwidth and storage requirements, not from processing requirements. For example, a chip in 3μ CMOS has been fabricated and tested for the wide-serial architecture in which about 4 percent of the area is used for processing. Any more processing on the chip would simply go unused because of storage and bandwidth constraints. We can expect this fraction to shrink as the lattice gets wider, and as we increase the dimensionality of the problems. This fact has recently become clear in the literature on systolic arrays, and in [5], Hong and Kung present a model and a bounding technique for quantifying this notion. In the next section, we will apply their results to the class of lattice computations.

7. Pebbling bounds

WSA and SPA are only two of many possible computation schemes for computing the evolution of a lattice gas cellular automaton (LGCA). Once a scheme has been selected from among the possibilities (for example, single stream pipeline, wide pipeline, column parallel), the processors and local memory must be mapped to chips while maintaining pin, area, processing rate, and I/O bandwidth constraints. These constraints can be thought of as divided into hierarchical classes by scale: main memory bandwidth, total processor memory, and overall computation rate at large scale; processing element area and speed at small scale; and inter-chip communication and pin constraints somewhere in between. The question arises as to which scheme makes the best use of the resources given the multi-scale constraints. To answer this partially, we would like to answer the general question, "What is the best that can be done, considering only the large scale constraints?" By "best" we mean "fastest overall computation rate." We want to ignore the particular method of progressing through the computation graph for a given LGCA and concentrate on the limits implied solely by the large scale constraints. We will use a pebble game to count the input/output requirements

of an LGCA computation.

Variants of the pebble game have been used as a tool to get space-time trade-offs for computational problems. See, for instance, the papers by Pippenger [11] and Lengauer and Tarjan [9]. The red-blue pebble game described by Hong and Kung [5] models the computation and I/O steps in a sequential computation. They used it to get space-input/output trade-offs for several problems, and to get upper bounds on speed-up of a computation of these problems using a sequential machine. The red-blue game they describe was extended by Savage and Vitter [15] to the parallel-red and block-red-blue pebble games, which model parallel computation without input/output and block parallel input/output respectively. We will use a further variant of the red-blue game which allows parallel computation and parallel input/output of any size up to the processor's local memory capacity. We will use Hong and Kung's methods for the analysis of the red-blue game to derive from our variant a trade-off among the minimum main memory bandwidth, the maximum overall computation rate, and the local processor memory size.

The red-blue pebble game is played on directed acyclic graphs with bounded in-degree according the following rules:

1. A pebble may be removed from a vertex at any time.
2. A red pebble may be placed on any vertex that has a blue pebble.
3. A blue pebble may be placed on any vertex that has a red pebble.
4. If all immediate predecessors of a vertex v are red pebbled, v may be red pebbled.

The "inputs" are those vertices which have no predecessors, and the "outputs" are those which have no successors. A vertex that is blue-pebbled represents the associated value's presence in main memory. A red-pebbled vertex represents presence in processor (chip) memory. Rules (2) and (3) represent I/O, and rule (4) represents the computation of a new value. The goal of the game is to blue-pebble the outputs given a starting configuration in which the inputs are blue-pebbled and the rest of the vertices are free of pebbles. We will delay the introduction of an extension of this game until we have established some further groundwork.

The computation graph for an LGCA is derived in the usual manner for a data dependency graph. An LGCA, $\mathcal{G} = G(v)$, is defined by a lattice graph $G = (V, E)$ contained in some d-dimensional finite volume, a value $v(x,\ t)$ associated with each node x in the lattice, and a function giving $v(x,\ t+1) = f(N(x), t)$ where $N(x)$ is the "neighborhood" of x in G; that is,

$$N(x)\{z|\{x,z\} \text{ is an edge in } G\} \cup \{x\}.$$

The values of nodes at time $t+1$ depend on the values of its neighboring nodes at time t. For an LGCA that models real fluids, the lattice G must be isotropic with respect to conservation of momentum and energy. This

means G must be regular. We will make use of this regularity in the proof for the bound on the computation rate, although we will not require the satisfaction of the isotropy condition. We form the computation graph of the LGCA by identifying the vertices in each layer of the computation graph with the vertices in the lattice G. Each layer of the computation graph consists of a copy of G's vertex set with arcs to the next layer expressing the data dependency between the values associated with the vertices of the lattice at time t and those at time $t+1$. That is, if $V = \{1, 2, 3, \ldots, L\}$ is the set of vertices in G, then the computation graph for G is $\mathcal{C} = (X, A)$ where

$$X = \{(x,t) | x \in V, \text{ and } 0 \leq t \leq T\}$$

and there is an arc from $(u, t-1)$ to (v, t) in $\mathcal{C}$ if and only if u is in $N(v)$. $\mathcal{C}$ is a layered graph of $T+1$ layers, each layer representing the LGCA at evolution time $t = 0, 1, 2, \ldots, T$ (see figures 5 and 6). We are usually interested in seeing an image of the LGCA at periodic time steps in its evolution, say every k time steps, and we let T go to infinity. However, it is easy to see from the proofs that follow that forcing $T = k$ will not alter the results. We will apply a variant of the red-blue pebble game to the computation graph $\mathcal{C}$.

Let us introduce some terms we will need and review the results of Hong and Kung. Results proved in [5] will be so indicated. A computation of an LGCA is said to be a *complete computation* if it begins with only the input values $v(x, 0)$ known and at the end of the computation the values $v(x, T)$ have been computed for all x in the lattice G of the LGCA. Thus, a pebbling P of the computation graph represents a complete computation of the LGCA. Given any complete computation of LGCA $\mathcal{G}$ (a pebbling P of the associated computation graph $\mathcal{C}_\mathcal{G}$, we assume the following, where memory and I/O are measured in units of storage required to store a single site value $v(x, t)$ of the LGCA.

S = the number of red pebbles, i.e., the amount of processor memory. (We assume an inexhaustible supply of blue pebbles.)

q = the number of I/O moves required by P.

Q = the minimum number of I/O moves required to pebble $\mathcal{C}$, over all pebblings using S or fewer red pebbles.

Definition: P' is an *S-I/O-division* of P, if

$$P' = \{P_i | 1 \leq i \leq h\}$$

where P_i is a consecutive subsequence of P such that P_i contains exactly q_i I/O moves, and

$$P = P_1 \circ P_2 \circ \cdots \circ P_h,$$

where

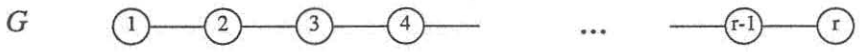


Figure 5: A one-dimensional lattice of a cellular automaton $\mathcal{G} = (G, v)$. Vertices 1 and r are boundary vertices of G. The neighborhood of vertex 2 is $N(2) = \{1, 2, 3\}$.

$q_i = S$ for all i except that $0 < q_h \leq S$.

We say the *size* of P' is h.

Clearly, a lower bound on the I/O required by a complete computation of $\mathcal{G}$ is determined by $\hat{h} = \min\{h\}$ over all pebblings of $\mathcal{C}_\mathcal{G}$ using S or fewer red pebbles. That is, $Q > S \cdot (\hat{h} - 1)$.

Hong and Kung have developed some methods for deriving a lower bound for $\hat{h}$. The concepts depend explicitly on the definition of an S-I/O-division which depends implicitly on the fact that the pebbling is linearly ordered. This is trivially true for the red-blue game because it is a strictly sequential game: a single rule from rules (1) through (4) is applied, and the resulting configuration determines the applicable rules for the next move. An immediate extension of the red-blue game simply considers a block of such moves as occurring in a single "time step". This allows a certain form of parallelism and is the extension used by Savage and Vitter [15] in the block-red-blue game. The actual play of the game is not altered; rather, the counting of moves is redefined. It is easy to find a simple example of a graph for which the number of input/output steps can be reduced by allowing the red pebbling moves to occur in truly parallel fashion. That is, any number of pebbles may be moved simultaneously, provided the configuration before the move satisfies the conditions of any rule employed in the move. With this in mind, we define the *parallel-red-blue* pebble game and show that it models any computation which can be performed by a computer with arbitrary parallel capabilities (CRCW PRAM).[3] The results of the analysis of this game will be applied to a machine model which has the same features as a CRCW PRAM, but has a limited communication bandwidth.

Consider a computation which proceeds by doing many steps in parallel in real time, and let us consider the necessary features of a pebble game that models it. The end result is a pebble game that can be described by a linearly ordered set of pebble moves, which will allow us to define an S-I/O-division for this game. In the following, we will use the following terminology: placing a red pebble on a node that contains no pebbles is a *calculation*. The node

[3]Such a machine model consists of an arbitrary number of processors communicating via a shared memory. This model is often referred to as a CRCW PRAM: Concurrent-Read Concurrent-Write Parallel Random Access Machine [2].

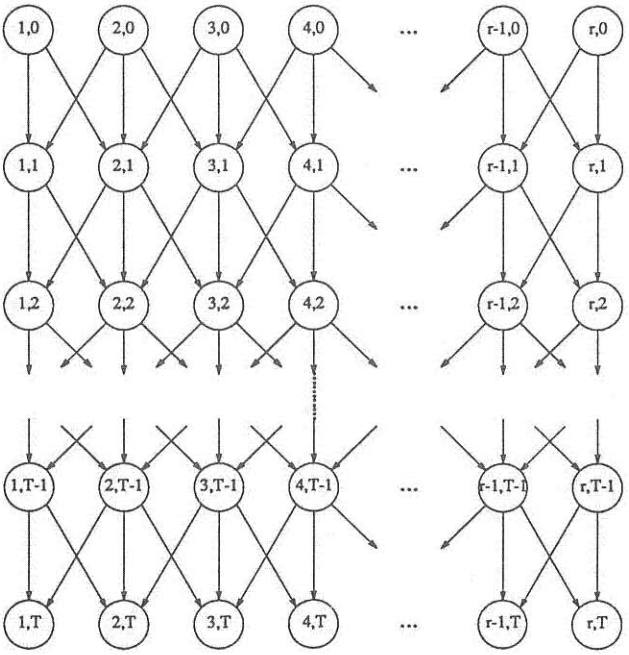


Figure 6: A computation graph $C_{\mathcal{G}}(T)$ where $0 \leq t \leq T$. The t^{th} row corresponds to $\mathcal{G}(t)$.

pebbled is called the *dependent* node, and the nodes with arcs ending at the dependent node are said to *support* the calculation by virtue of the fact that if they did not contain red pebbles, the calculation would not be possible.

We first decompose the computation into pieces which occur simultaneously. Let these pieces be designated C_i, and we say the complete computation C consists of their concatenation: $C = C_1 \circ C_2 \circ \cdots \circ C_k$. Now let us consider the pebble moves within C_i. Consider a datum that is fetched from main memory by C_i. It is reasonable to assume that this datum could not simultaneously be used in a calculation of some dependent datum. We then require that a pebble move that places a red pebble on a node which only contains a blue pebble precedes any pebble move that uses the node as a supporting node for a calculation. We satisfy this ordering requirement by ordering all the pebble moves of this type (which model main memory reads occurring in C_i) after any other moves in C_i.

Consider the calculation of a datum during C_i. We assume the result datum must be written to a register in the processor memory. Therefore, we do not consider it possible in our model of computation to allow a main memory write of a datum to occur simultaneously with a calculation of the same datum. We can enforce this requirement in the pebble game by ordering

all main memory writes in C_i before all calculations in C_i. That is, a node must contain a red pebble before a blue pebble may be placed on it, and that red pebble must have been placed in a previous C_i.

At this point, we can say that the pebble game must proceed parallel move by parallel move in order and that within each parallel move C_i the ordering is: place blue pebbles (write to main memory phase), move red pebbles to unpebbled nodes (calculation phase), place red pebbles on nodes containing blue pebbles (read-from-main-memory phase). It now remains for us to find an ordering within these phases of C_i.

Consider the pebble moves in the two I/O phases. In real time, we assume they all happen simultaneously. Suppose we order them arbitrarily within each phase. Take first the write phase. Placing blue pebbles on nodes containing red pebbles in any order is permissible since there are no dependence constraints beyond the presence of the red pebbles. The nodes available for writing have red pebbles before the beginning of C_i.

The read phase is essentially the same, except that nodes containing blue pebbles receive red pebbles. We must be careful not to violate the timing constraints on any red pebbles used for this purpose. A register that is used to store the result of a calculation performed during C_i cannot also receive a datum from main memory during C_i. Actually, the red pebbles placed on dependent nodes in the calculation phase could theoretically be picked up and moved to a blue-pebbled node to effect a read during the read phase. However, the result is that the dependent node that had its red pebble removed did not really get calculated during C_i, and we are not violating the timing constraints on the real-time computation if we adopt this interpretation of such an event.

The next potential conflict comes from the overlapping of read and write operations. Suppose a register is used as a source to write to main memory. During the write phase, a blue pebble is placed on a node containing a red pebble. The red pebble represents the use of a register as a source for a write operation. However, the read phase may remove the red pebble and place it on some node containing a blue pebble, indicating the same register is both a source and a receiver of data simultaneously. We accept this as within our model of computation because hardware with this capability is easily realized. The only remaining sources for red pebbles are red pebbles that were placed prior to C_i and do not therefore represent any conflict with real computation. As the various sources for red pebbles do not have any mutual dependencies, and likewise, the placement of the red pebbles are not inter-dependent; we are free to order the movement of pebbles in the read phase arbitrarily.

We have established that the I/O phases may be linearly ordered. At this point, it appears that nothing has changed vis-a-vis the red-blue game. The real contribution of the parallel game comes in the calculation phase. Consider a calculation in which the result is written into one of the registers used as input. The input may be fanned out to many calculations, and all proceed in parallel. The red-blue game would block this type of activity

since lifting the red pebble from a supporting node and sliding it to one of the dependent nodes leaves the remaining dependent nodes without a full complement of supporting nodes. We define the calculation phase as the movement of red pebbles onto dependent nodes. We will add a new pebble (pink) to the game to avoid the blockage mentioned above. The pink pebble (place-holder pebble) allows fan-out of the input by holding the contents of the calculation until the end of the calculation phase. The new pebble is not strictly required, but using it simplifies the definition of the new game.

The above discussion gives us a pebble game that can model an arbitrary parallel computation under the assumed model of computation. The game is sequential in the I/O phases, and taking the calculation phase as a single move, the S-I/O-division is well defined for this game.

Definition: The rules of the *parallel-red-blue* pebble game:
The game is identical to the red-blue pebble game with the addition of a new pebble (pink) and the following additional rules:

5. The game consists of cyclic repetition of three phases:
   write phase, calculate phase, read phase.
   The write phase consists of only rule (3) moves.
   The read phase consists of only rule (2) moves.
   The calculate phase comprises the following moves
   (a) pink pebble placed by rule (4).
   (b) a red pebble replaces a pink pebble.
   (c) no pink pebbles remain at the end of the phase.

With this definition of the parallel-red-blue game, we can proceed along the lines of [5] without altering their arguments. Their next step introduces the idea of partitioning the computation graph to get a lower bound on the number of sub-pebblings in an S-I/O-division.

Definition: A *K-partition* $\mathcal{V}$ is a partition of the vertices of a directed acyclic graph $G = (V, A)$ such that

1. For every V_i in $\mathcal{V}$ there is a *dominator set* $D_i \subseteq V$, and a *minimum set* $M_i \subseteq V_i$, both of size at most K such that every path from the inputs to any element of V_i contains an element of D_i, and every v in V_i which has no children in V_i is in M_i.

2. There are no cyclic dependencies among the V_i. (V_j *depends* on V_i if there is an arc from an element of V_i to some element of V_j.)

We say $g = |\mathcal{V}|$ is the *size* of the partition.

For every S-I/O-division of a pebbling P there is a $2S$-partition determined in the following way: in P, consider every vertex that has never had a red pebble placed on it by any moves in P_i, $i < k$, and is red pebbled during P_k. This set of vertices is V_k. Property (2) is clearly satisfied by the set all

such V_k's, $\mathcal{V}$. The dominator, D_k, is then the set of all vertices which had red pebbles on them at the end of P_{k-1}, together with those vertices with blue pebbles on them at the end of P_{k-1} which get red pebbles during P_k. The size of D_k is at most $2S$ (there are S red pebbles and at most S I/O moves). The minimum set, M_k, consists of those vertices which were the "last" to be red pebbled during P_k (i.e., have no children which were red pebbled during P_k). At the end of P_k, any such vertex is either i) still red pebbled, or ii) now blue pebbled. Therefore, M_k can be at most of size $2S$.

The above argument gives us the following theorem and lemma.

Theorem 2. *[5] Let G be any directed acyclic graph and P be any red-blue pebbling of G with an S-I/O-division of size h using at most S red pebbles. Then, there is a $2S$-partition of G of size $g = h$.*

In particular, there is a partition such that $g = \hat{h}$. From the comment made above concerning the minimum I/O requirements, and letting $\hat{g} = \min\{g\}$ over all $2S$-partitions of G, we have

Lemma 1. *[5] For any directed acyclic graph,*

$$Q > S \cdot (\hat{g} - 1).$$

The types of graphs represented by LGCA computation graphs have the nice feature that they are regular and "lined." *Lines* are simple paths from inputs to outputs. A vertex is said to *lie on a line* if the line contains the vertex. A line is *covered* by a set of vertices if the set contains a vertex that lies on the line. A *lined graph* is a graph in which a set of vertex disjoint lines can be chosen so that every input is on some line in the set. A *complete set of lines* is such a set of lines. For an LGCA computation graph, a path $\langle (x,0),(x,1),(x,2),\ldots,(x,T)\rangle$ is a line l_x, for any node x in the lattice G. Suppose we have chosen a complete set of lines $\mathcal{L}$ for some lined graph G. If we can bound from above the maximum number of vertices that lie on lines in $\mathcal{L}$ and are contained in a single subset of any $2S$-partition of G, and we can count the total number of vertices in G that are on lines, we will be able to lower bound $\hat{g}$. In applying this reasoning to LGCA computation graphs, we will choose the complete set of lines

$$\mathcal{L} = \{l_x | x \in \mathrm{V}\}.$$

In the case of these graphs, every vertex lies on some line in $\mathcal{L}$.

Definition: The *line-time* $\tau(k)$ for a lined graph G is the maximum number of vertices that lie on a single line in any subset of any k-partition of G. That is, if we let $\mathcal{K}$ be the set of all k-partitions of G and $\mathcal{L}$ be a complete set of lines in G, then

$$\tau(k) = \max_{\mathcal{V}\in\mathcal{K}} \max_{V_i\in\mathcal{V}} \max_{l_i\in\mathcal{L}} \{|l_i \cap V_i|\}.$$

By observing that a dominator set of size $2S$ or less can dominate at most $2S$ different lines, it is easy to conclude that the maximum number of vertices in a single subset of a $2S$-partition that lie on lines is bounded from above by $2S \cdot \tau(2S)$; that is,

$$|V_i^*| \leq 2S \cdot \tau(2S) \text{ in any } 2S\text{-partition of } G,$$

where V_i^* is the smallest subset of V_i containing every vertex in V_i that lies on some line.

Consequently, we have

Lemma 2. *[5]* $\hat{g} \geq \frac{|X^*|}{2S \cdot \tau(2S)}$ *for a computation graph* $C = (X, A)$.

This leads to Hong and Kung's second result:

Theorem 3. *[5]* $Q = \Omega(\frac{|X^*|}{\tau(2S)})$.

For LGCA computations, we can express this bound in terms of main memory bandwidth B and processing rate R. Let the total time to perform the computation described by the computation graph be p. We then define $R = \frac{|X|}{p}$ (for LGCA computations $|X| = |X^*|$). Certainly, the total input/output traffic must be handled by the communication channel to main memory, so $Bp \geq Q$, and the preceding bound becomes

$$B = \Omega(\frac{R}{\tau(2S)})$$

or equivalently,

$$R = O(B\tau(2S)).$$

Using this result, we will show that for d-dimensional LGCA computations

$$R = O(BS^{\frac{1}{d}}).$$

Specifically, we will show that

$$\tau(2S) < 2(d!2S)^{\frac{1}{d}}$$

for their computation graphs.

In proving this, we will make the following simplifying assumptions, which are in any event worst-case.

1. The graph G of a d-dimensional LGCA is an orthogonal grid defined on the integer lattice points contained in the d-cell in $\mathbf{R}^d$ consisting of the points $\{\mathbf{x} | 0 \leq x_i \leq r (i = 1, 2, \ldots, d)\}$ where r is a non-negative integer. There are edges between a vertex and its nearest neighbors. We will refer to G as a lattice. Although G as defined above is inadequate for isotropic lattice gases [3], we are assuming the minimum connectivity for G in the sense that any lattice that satisfies isotropy requires at least the same degree of connectivity.

2. The boundaries of LGCAs can be handled in a variety of ways. They can be null (zero valued), independently random, dependently random or deterministic with truncated neighborhoods, or toroidally connected with full connectivity. In the first two cases above, the boundaries do not appear in $\mathcal{C}$ at all. We will assume boundary vertices appear in $\mathcal{C}$ with dependencies defined by the lattice. The boundaries can be thought of as deterministic or randomized, but dependent on their neighbors as defined in (1).
3. If the size of the vertex set of the lattice G is r^d, then we assume that the processor memory size S is less than r^d. In fact, if $S = r^d$, only $2S$ of main memory I/O is required to pebble $\mathcal{C}$, and the bounds mentioned are irrelevant.
4. In the following, we will use the notation $\mathcal{C}_d$ when referring to a computation graph $\mathcal{C}_\mathcal{G}$ for a d-dimensional LGCA $\mathcal{G}$, with lattice G.

Let us derive some properties of the computation graph $\mathcal{C}_d$.

Definition: A (u, v)-*path* is a path from vertex u to vertex v. The *length* of path p, $l(p)$, is the number of edges in p. The *distance* $d(u, v)$ between two vertices u and v is the minimum of $l(p)$ over all (u, v)-paths p.

Lemma 3. *In $\mathcal{C}_d$, every (u, v)-path p has length $d(u, v)$.*

Proof. Since every arc in $\mathcal{C}_d$ goes from some layer t to a layer $t + 1$, paths of different lengths starting from the same vertex end in different layers. ∎

Lemma 4. *In $\mathcal{C}_d$ every vertex w which has a distance from some specified vertex u of $d(u, w) = \frac{1}{2}\lfloor d(u, v) \rfloor$ lies on some (u, v)-path, provided u and v both lie on the same line in $\mathcal{L}$.*

Proof. Let $d(u, v) = 2k + \delta$ where $k \geq 0$ and δ is either zero or one. Let $u = (x, t)$ and $v = (x, t + 2k + \delta)$. If $k = 0$, the result is trivial, so suppose that $k > 0$. There is some (u, w)-path $p_1 = (u = u_0, u_1, \ldots, u_k = w)$. Let $u_i = (x_i, t + i)$. Since there is an arc (u_i, u_{i+1}), x_i is in the neighborhood $N(x_{i+1})$, and vice versa. Consequently, there is a path $p_2 = (w = v_k, v_{k-1}, \ldots, v_0 = (x, t+2k))$, where $v_i = (x_i, t+k+(k-i))$. Thus, the path $p = p_1 \circ p_2$ is a path from u to $(x,\ t + 2k)$ on the (u, v)-path along l_x. Concatenating the path $((x, t + 2k), (x, t + 2k + 1), (x, t + 2k + 2), \ldots, (x, t + 2k + \delta))$ onto the end of p gives us a (u, v)-path containing w. ∎

Lemma 5. *In $\mathcal{C}_d$, every line l_x covered by a path of length at most j from some specified vertex u, is covered by a path of length exactly j such that the last vertex on the path lies on l_x.*

Proof. Let p be a path from u of length j or less covering line l_x. Let z be a vertex on path p such that z lies on l_x. Let p_1 be that portion of p from u to z. By assumption $l(p_1) = k \leq j$. Concatenating onto p_1 the path starting from z and continuing along l_x for $j - k$ steps gives us the required path. ∎

Lemma 6. *In $\mathcal{C}_d$ the number of lines covered by all paths of length j or less from a specified vertex u is equal to the number of vertices reachable from u in exactly j steps.*

Proof. By the definition of $\mathcal{L}$, every vertex in a single layer lies on a unique line. By the argument of lemma 3, the end point of every path of length j lies in the same layer. So, for every vertex reachable in exactly j steps, there is a line covered by a path of length j. The lemma then follows from the previous lemma. ∎

Lemma 7. *If in $\mathcal{C}_d$ vertex $v = (z, t+j)$ is reachable from vertex $u = (x, t)$ in j steps, then in G vertex z is reachable from x in at most j steps. The converse holds if $t \leq T - j$.*

Proof. Consider a (u, v)-path $p = (u = u_0, u_1, u_2, \ldots, u_j = v)$ in $\mathcal{C}_d$, where $u_i = (x_i, t+i)$. Since $x_i \in N(x_{i+1})$, either there is an edge $\{x_i, x_{i+1}\}$ in G, or $x_i = x_{i+1}$. Deleting the self loops from the path $q = (x = x_0, x_1, x_2, \ldots, x_j = z)$ gives us an $(x,\ z)$-path in G of length at most j. ∎

Conversely, consider a path $q = (x_0 = x, x_1, x_2, \ldots, x_i = z)$ in G where $0 \leq i \leq j$. By hypothesis, $t \leq T - j$, and consequently, the path

$$p = ((x = x_0, t), (x_1, t+1), \ldots, (x_i = z, t+i), (z, t+i+1), \ldots, (z, t+j))$$

is a (u, v)-path in $\mathcal{C}_d$.

Definition: The *line-spread from a vertex u* in graph G is

$$t_G(u, j) = \begin{cases} \infty, \text{ if no vertex } z \text{ exists such that } d(u, z) = j \\ \text{the number of lines covered} \\ \text{by paths of length } j \text{ or less, otherwise.} \end{cases}$$

Definition: The *line-spread* of a graph $G = (V, E)$ is

$$T_G(j) = \min_{u \in V} \{t_G(u, j)\}$$

If the graph G is $\mathcal{C}_d$, we write $T_d(j)$.

Lemma 8. $T_d(j) > \frac{j^d}{d!}$.

Proof. By lemmas 5, 6, and 7 we have shown that the number of lines covered by paths from some vertex $u = (x, t)$ of length at most j in $\mathcal{C}_d$ is equal to the number of vertices reachable from x in at most j steps in G, provided at least one path of length j exists in $\mathcal{C}_d$. By the definition of G, that is, an integer grid in the non-negative orthant, the minimum number of vertices reachable in j or fewer steps in G occurs when the origin is chosen as the specified vertex. The latter quantity is then given by

$$T_d(j) = \sum_{i=0}^{j} T_{d-1}(i) = \binom{j+d}{j} = \sum_{\Phi} 1 > \int_{\phi} dx = \frac{j^d}{d!}$$

where ϕ is the region in $\mathbf{R}^d$ defined by the set $\{\mathbf{x}|x_1 + x_3 + \cdots + x_d \leq j, (x_i \geq 0)\}$, and Φ is the set of integer lattice points in ϕ.

We are now in a position to prove the main result:

Theorem 4. *In $\mathcal{C}_d$, $\tau(2S) < 2(d!2S)^{\frac{1}{d}}$.*

Proof.[4] Suppose that $\tau(2S) \geq 2(d!2S)^{\frac{1}{d}}$. Let $j = (d!2S)^{\frac{1}{d}}$. Then there exist vertices u and v in some subset V_i of some $2S$-partition $\mathcal{V}$ of $\mathcal{C}_d$ such that $d(u,v) = 2j$, and u and v both lie on the same line in $\mathcal{L}$. Since the subsets of the partition $\mathcal{V}$ are not cyclically dependent, every vertex z on any (u,v)-path is in V_i. By lemma 4, every vertex in the set $Z = \{z|d(u,z) = j\}$ is on a (u,v)-path, and therefore $Z \subset V_i$. Then Z covers at least $T_d(j)$ lines. The dominator for V_i must cover these lines. Since the lines in $\mathcal{L}$ are disjoint $|D_i| \geq T_d(j)$, and employing lemma 8 we have $|D_i| > \frac{j^d}{d!} = 2S$. This contradicts the assumption that V_i is an element of a $2S$-partition, and we are done. ∎

8. Conclusions

We have described two architectures for lattice-update engines based on VLSI custom chips and derived their design curves and best operating points. The wide-serial architecture (WSA) has extremely simple support logic and data flow, while Sternberg's partitioned architecture (SPA) is perhaps more easily extensible to lattices of arbitrary sizes and provides higher throughput per custom chip, albeit at the expense of support logic and main memory bandwidth. Each has its preferred operating regime in different parts of the throughput vs. lattice-size plane. A prototype lattice-gas engine, using the WSA architecture, and based on a custom 3μ CMOS chip, is now being constructed. Each chip provides 20 million site-updates per second running at 10 MHz. It is unlikely, however, that the workstation host will be able to supply the 40 megabyte per second bandwidth required for this level of performance. We expect to realize approximately 1 million site-updates/sec/chip from the prototype implementation.

We have also presented a graph-pebbling argument that gives upper bounds for the computation rate for lattice updates. The asymptotic upper bounds show clearly that memory bandwidth, and not processor speed or size, is the factor that limits performance. One goal for further research is the tightening of these pebbling-game arguments so that they give estimates of absolute, as well as asymptotic, performance. We will apply these estimates to get quantitative comparisons between competing architectures

[4]This proof follows the proof of theorem 5.1 in [5].

for lattice gas computations such as the Connection Machine, the CRAY-XMP, and special purpose machines. A further goal would be to discover an optimal pebbling for any problem in this class, and thereby discover an architecture which is optimal with regard to input/output complexity.

This work supports the growing recognition that communication bottlenecks—at all scales of the architectural hierarchy—are the critical limiting factors in the performance of highly pipelined, massively parallel machines. In our conservative VLSI design, not nearly at the limits of present integration technology, the processors themselves comprise only a small fraction of the total silicon area. As feature sizes shrink and problems are tackled with larger lattices in higher dimensions, this effect will become even more dramatic. This suggests that a search for more effective interconnection technologies, perhaps using optics, should have high priority.

References

[1] D. d'Humiéres, P. Lallemand, and U. Frisch, "Lattice Gas Models for 3D Hydrodynamics," *Europhysics Letters*, **4:2** (1986).

[2] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proc. 10th Annual ACM Symp. on the Theory of Computing*, San Diego, CA, 1978.

[3] U. Frisch, B. Hasslacher, and Y. Pomeau, "A Lattice Gas Automaton for the Navier-Stokes Equation," Los Alamos National Laboratory preprint LA-UR-85-3503 (1985).

[4] J. Hardy, Y. Pomeau, and O. de Pazzis, "Time evolution of a two-dimensional model system. I. Invariant states and time correlation functions," *J. Math. Phys.*, **14:12** (1973) 1746–1759.

[5] Jia-Wei Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proceedings of ACM Sym. Theory of Computing*, (1981) 326–333.

[6] Josef Kittler and Michael J. B. Duff, eds. *Image Processing System Architectures*, (Research Studies Press, Ltd., John Wiley and Sons, 1985).

[7] Steven D. Kugelmass, Richard Squier, and Kenneth Steiglitz, "Performance of VLSI Engines for Lattice Computations," *Proc. 1987 Int. Conf. on Parallel Processing*, St. Charles, IL, August 17–21, (Pennsylvania State University Press, University Park, PA, 1987) 684–691.

[8] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront Array Processors—Concept to Implementation," *Computer*, **20:7** (IEEE, New York, 1987).

[9] T. Lengauer and R. E. Tarjan, "Upper and Lower Bounds on Time-Space Tradeoffs," *ACM Symposium on the Theory of Computing*, Atlanta, GA (1979) 262–277.

[10] Steven A. Orszag and Victor Yakhot, "Reynolds Number Scaling of Cellular Automaton Hydrodynamics," *Physical Review Letters*, **56:16** (1986) 1691–1693.

[11] N. Pippenger, "Pebbling," *Proc. 5th IBM Symposium on Mathematical Foundations of Computer Science*, Academic and Scientific Programs, IBM Japan, May 1980.

[12] Arnold L. Rosenberg, "Preserving Proximity in Arrays," *SIAM J. Computing*, **4:4** (1975) 443–460.

[13] Peter A. Ruetz and Robert W. Broderson, "Architectures and Design Techniques for Real-Time Image-Processing IC's," *IEEE Journal of Solid-State Circuits*, **SC-22:2** (1987).

[14] James B. Salem and Stephén Wolfram, "Thermodynamics and Hydrodynamics with Cellular Automata," in *Theory and Applications of Cellular Automata*, ed. S. Wolfram, (World Scientific Publishing Co., Hong Kong, 1986) 362–366.

[15] John E. Savage and Jeffrey Scott Vitter, "Parallelism in Space-Time Tradeoffs," in *VLSI: Algorithms and Architectures*, ed. F. Luccio, (Elsevier Science Publishers B.V., North Holland, 1985) 49–58.

[16] K. Steiglitz and R. R. Morita, "A Multi-Processor Cellular Automaton Chip," *Proc. 1985 IEEE Int. Conf. on Acoustics, Speec, and Signal Processing*, Tampa, FL, 1985.

[17] Stanley R. Sternberg, "Computer Architectures Specialized for Mathematical Morphology," in *Algorithmically Specialized Parallel Computers*, ed. Howard Jay Siegel, (Academic Press, 1985) 169–176.

[18] Stanley R. Sternberg, "Pipeline Architectures for Image Processing," in *Multicomputers and Image Processing, Algorithms and Programs*, ed. Leonard Uhr, (Academic Press, 1982) 291–305.

[19] Kenneth Supowit and Neal Young, personal communication (1986).