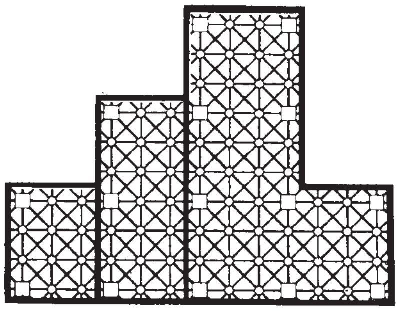
Systolic architectures, which permit multiple computations for each memory access, can speed execution of compute-bound problems without increasing I/O requirements.



Why Systolic Architectures?

H. T. Kung
Carnegie-Mellon University

High-performance, special-purpose computer systems are typically used to meet specific application requirements or to off-load computations that are especially taxing to general-purpose computers. As hardware cost and size continue to drop and processing requirements become well-understood in areas such as signal and image processing, more special-purpose systems are being constructed. However, since most of these systems are built on an ad hoc basis for specific tasks, methodological work in this area is rare. Because the knowledge gained from individual experiences is neither accumulated nor properly organized, the same errors are repeated. I/O and computation imbalance is a notable example—often, the fact that I/O interfaces cannot keep up with device speed is discovered only after constructing a high-speed, special-purpose device.

We intend to help correct this ad hoc approach by providing a general guideline—specifically, the concept of systolic architecture, a general methodology for mapping high-level computations into hardware structures. In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. The system works like an automobile assembly line where different people work on the same car at different times and many cars are assembled simultaneously. An assembly line is always linear, however, and systolic systems are sometimes two-dimensional. They can be rectangular, triangular, or hexagonal to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple directions—both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. Generally speaking, a systolic system is easy to implement because of its regularity and easy to reconfigure (to meet various outside constraints) because of its modularity.

The systolic architectural concept was developed at Carnegie-Mellon University,[1-7] and versions of systolic processors are being designed and built by several industrial and governmental organizations.[8-10] This article reviews the basic principle of systolic architectures and explains why they should result in cost-effective, high-performance special-purpose systems for a wide range of problems.

Key architectural issues in designing special-purpose systems

Roughly, the cycle for developing a special-purpose system can be divided into three phases—task definition, design, and implementation. During task definition, some system performance bottleneck is identified, and a decision on whether or not to resolve it with special-purpose hardware is made. The evaluation required for task definition is most fundamental, but since it is often application-dependent, we will concentrate only on architectural issues related to the design phase and will assume routine implementation.

Simple and regular design. Cost-effectiveness has always been a chief concern in designing special-purpose systems; their cost must be low enough to justify their limited applicability. Costs can be classified as nonrecurring (design) and recurring (parts) costs. Part costs are dropping rapidly due to advances in integrated-circuit technology, but this advantage applies equally to both special-purpose and general-purpose systems. Furthermore, since special-purpose systems are seldom produced in large quantities, part costs are less important than design costs. Hence, the design cost of a special-purpose system must be relatively small for it to be more attractive than a general-purpose approach.

Fortunately, special-purpose design costs can be reduced by the use of appropriate architectures. If a structure can truly be decomposed into a few types of simple substructures or building blocks, which are used repetitively with simple interfaces, great savings can be achieved. This is especially true for VLSI designs where a single chip comprises hundreds of thousands of components. To cope with that complexity, simple and regular designs, similar to some of the techniques used in constructing large software systems, are essential.[11] In addition, special-purpose systems based on simple, regular designs are likely to be modular and therefore adjustable to various performance goals—that is, system cost can be made proportional to the performance required. This suggests that meeting the architectural challenge for simple, regular designs yields cost-effective special-purpose systems.

Concurrency and communication. There are essentially two ways to build a fast computer system. One is to use fast components, and the other is to use concurrency. The last decade has seen an order of magnitude decrease in the cost and size of computer components but only an incremental increase in component speed.[12] With current technology, tens of thousands of gates can be put in a single chip, but no gate is much faster than its TTL counterpart of 10 years ago. Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements. The degree of concurrency in a special-purpose system is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant—especially with VLSI technology where routing costs dominate the power, time, and area required to implement a computation.[13] The issue here is to design algorithms that support high degrees of concurrency, and in the meantime to employ only simple, regular communication and control to enable efficient implementation.

Balancing computation with I/O. Since a special-purpose system typically receives data and outputs results through an attached host, I/O considerations influence overall performance. (The host in this context can mean a computer, a memory, a real-time device, etc. In practice, the special-purpose system may actually input from one "physical" host and output to another.) The ultimate performance goal of a special-purpose system is—and should be no more than—a computation rate that balances the available I/O bandwidth with the host. Since an accurate a priori estimate of available I/O bandwidth in a complex system is usually impossible, the design of a special-purpose system should be modular so that its structure can be easily adjusted to match a variety of I/O bandwidths.

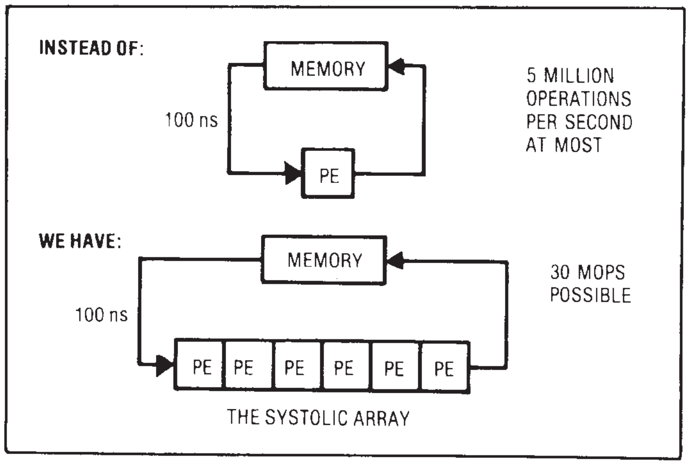


Figure 1. Basic principle of a systolic system.

Suppose that the I/O bandwidth between the host and a special-purpose system is 10 million bytes per second, a rather high bandwidth for present technology. Assuming that at least two bytes are read from or written to the host for each operation, the maximum rate will be only 5 million operations per second, no matter how fast the special-purpose system can operate (see Figure 1). Orders of magnitude improvements on this throughput are possible only if multiple computations are performed per I/O access. However, the repetitive use of a data item requires it to be stored inside the system for a sufficient length of time. Thus, the I/O problem is related not only to the available I/O bandwidth, but also to the available memory internal to the system. The question then is how to arrange a computation together with an appropriate memory structure so that computation time is balanced with I/O time.

The I/O problem becomes especially severe when a large computation is performed on a small special-purpose system. In this case, the computation must be decomposed. Executing subcomputations one at a time may require a substantial amount of I/O to store or retrieve intermediate results. Consider, for example, performing the n-point fast Fourier transform using an S-point device when n is large and S is small. Figure 2 depicts the n-point FFT computation and a decomposition scheme for $n = 16$ and $S = 4$. Note that each subcomputation block is sufficiently small so that it can be handled by the 4-point device. During execution, results of a block must be temporarily sent to the host and later retrieved to be combined with results of other blocks as they become available. With the decomposition scheme shown in Figure 2b, the total number of I/O operations is $O(n \log n/\log S)$. In fact, it has been shown that, to perform the n-point FFT with a device of $O(S)$ memory, at least this many I/O operations are needed for any decomposition scheme.[14] Thus, for the n-point FFT problem, an S-point device cannot achieve more than an $O(\log S)$ speed-up ratio over the conventional $O(n \log n)$ software implementation time, and since it is a consequence of the I/O consideration, this upper bound holds independently of device speed. Similar upper bounds have been established for speed-up ratios achievable by devices for other computations such as sorting and matrix multiplication.[14,15] Knowing the I/O-imposed performance limit helps prevent overkill in the design of a special-purpose device.

In practice, problems are typically "larger" than special-purpose devices. Therefore, questions such as how a computation can be decomposed to minimize I/O, how the I/O requirement is related to the size of a special-purpose system and its memory, and how the I/O bandwidth limits the speed-up ratio achievable by a special-purpose system present another set of challenges to the system architect.

Systolic architectures: the basic principle

As a solution to the above challenges, we introduce systolic architectures, an architectural concept originally proposed for VLSI implementation of some matrix operations.[5] Examples of systolic architectures follow in the next section, which contains a walk-through of a family of designs for the convolution computation.

A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells." For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system.

Computational tasks can be conceptually classified into two families—compute-bound computations and I/O-bound computations. In a computation, if the total number of operations is larger than the total number of input and output elements, then the computation is compute-bound, otherwise it is I/O-bound. For example, the ordinary matrix-matrix multiplication algorithm represents a compute-bound task, since every entry in a matrix is multiplied by all entries in some row or column of the other matrix. Adding two matrices, on the other hand, is I/O-bound, since the total number of adds is not larger than the total number of entries in the two matrices. It should be clear that any attempt to speed up an I/O-bound computation must rely on an increase in memory bandwidth. Memory bandwidth can be increased by the use of either fast components (which could be expensive) or interleaved memories (which could create complicated memory management problems). Speeding up a compute-bound computation, however, may often be accomplished in a relatively simple and inexpensive manner, that is, by the systolic approach.

The basic principle of a systolic architecture, a systolic array in particular, is illustrated in Figure 1. By replacing a single processing element with an array of PEs, or cells in the terminology of this article, a higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the diagram is analogous to that of the heart; it "pulses" data (instead of blood) through the array of cells. The crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes while being "pumped" from cell to cell along the array. This is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner.

Being able to use each input data item a number of times (and thus achieving high computation throughput with only modest memory bandwidth) is just one of the many advantages of the systolic approach. Other advantages, such as modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting, and fan-in and (possibly) fast response time, will be illustrated in various systolic designs in the next section.

A family of systolic designs for the convolution computation

To provide concrete examples of various systolic structures, this section presents a family of systolic designs for the convolution problem, which is defined as follows:

Given the sequence of weights $\{w_1, w_2, \ldots, w_k\}$ and the input sequence $\{x_1, x_2, \ldots, x_n\}$,

compute the result sequence $\{y_1, y_2, \ldots, y_{n+1-k}\}$ defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

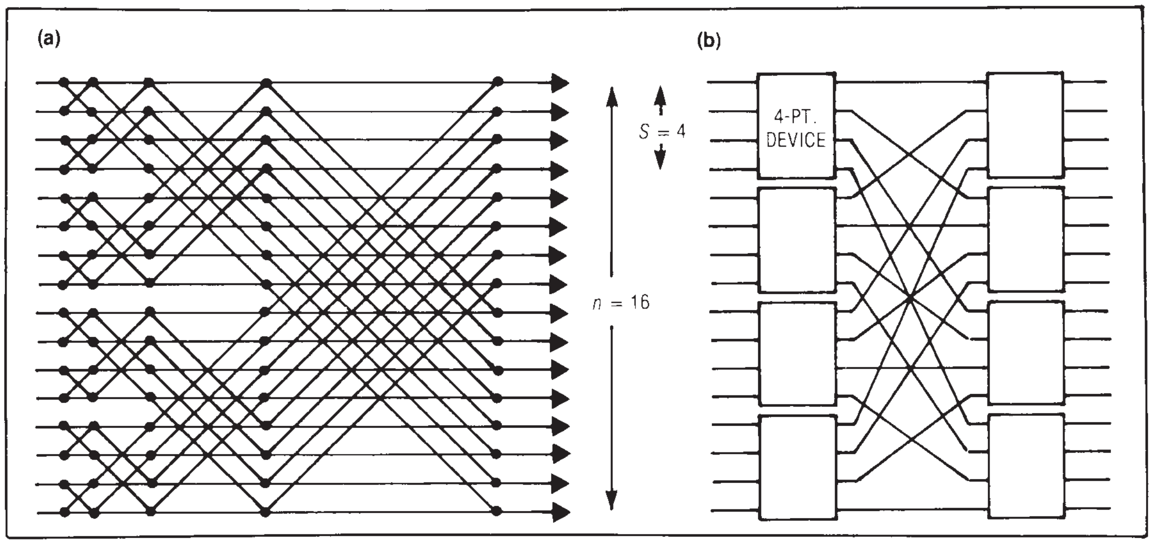


Figure 2. (a) 16-point fast-Fourier-transform graph; (b) decomposing the FFT computation with $n = 16$ and $S = 4$.

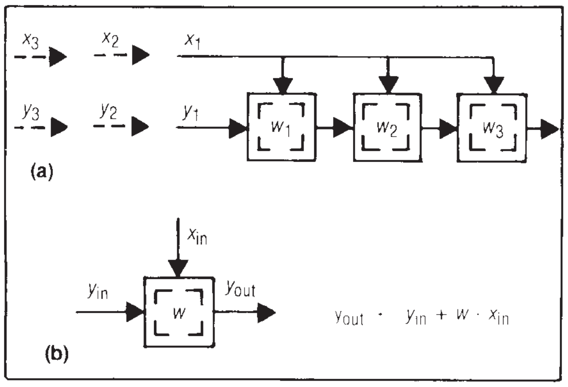


Figure 3. Design B1: systolic convolution array (a) and cell (b) where x_i's are broadcast, w_i's stay, and y_i's move systolically.

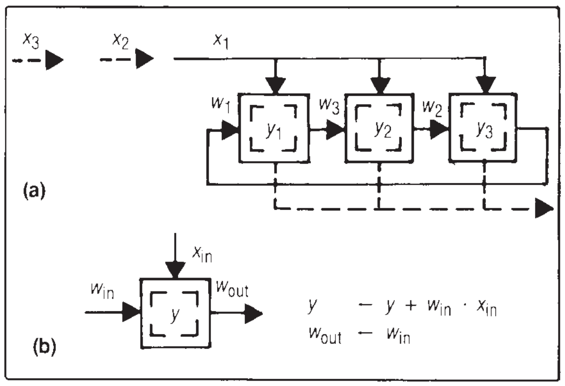


Figure 4. Design B2: systolic convolution array (a) and cell (b) where x_i's are broadcast, y_i's stay, and w_i's move systolically.

We consider the convolution problem because it is a simple problem with a variety of enlightening systolic solutions, because it is an important problem in its own right, and more importantly, because it is representative of a wide class of computations suited to systolic designs. The convolution problem can be viewed as a problem of combining two data streams, w_i's amd x_i's, in a certain manner (for example, as in the above equation) to form a resultant data stream of y_i's. This type of computation is common to a number of computation routines, such as filtering, pattern matching, correlation, interpolation, polynomial evaluation (including discrete Fourier transforms), and polynomial multiplication and division. For example, if multiplication and addition are interpreted as comparison and boolean AND, respectively, then the convolution problem becomes the pattern matching problem.[1] Architectural concepts for the convolution problem can thus be applied to these other problems as well.

The convolution problem is compute-bound, since each input x_i is to be multiplied by each of the k weights. If the x_i is input separately from memory for each multiplication, then when k is large, memory bandwidth becomes a bottleneck, precluding a high-performance solution. As indicated earlier, a systolic architecture resolves this I/O bottleneck by making multiple use of each x_i fetched from the memory. Based on this principle, several systolic designs for solving the convolution problem are described below. For simplicity, all illustrations assume that $k = 3$.

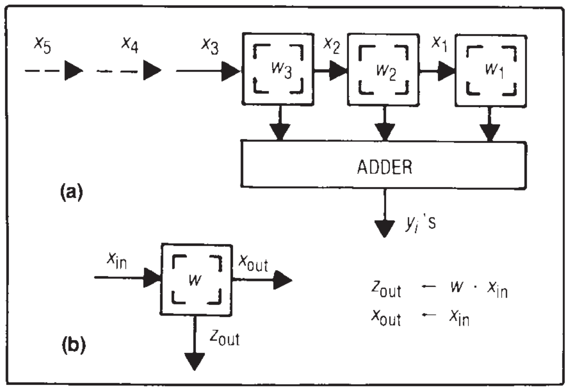
(Semi-) systolic convolution arrays with global data communication. If an x_i, once brought out from the memory, is broadcast to a number of cells, then the same x_i can be used by all the cells. This broadcasting technique is probably one of the most obvious ways to make multiple use of each input element. The opposite of broadcasting is fan-in, through which data items from a number of cells can be collected. The fan-in technique can also be used in a straightforward manner to resolve the I/O bottleneck problem. In the following, we describe systolic designs that utilize broadcasting and fan-in.

Design B1—broadcast inputs, move results, weights stay. The systolic array and its cell definition are depicted in Figure 3. Weights are preloaded to the cells, one at each cell, and stay at the cells throughout the computation. Partial results y_i move systolically from cell to cell in the left-to-right direction, that is, each of them moves over the cell to its right during each cycle. At the beginning of a cycle, one x_i is broadcast to all the cells and one y_i, initialized as zero, enters the left-most cell. During cycle one, $w_1 x_1$ is accumulated to y_1 at the left-most cell, and during cycle two, $w_1 x_2$ and $w_2 x_2$ are accumulated to y_2 and y_1 at the left-most and middle cells, respectively. Starting from cycle three, the final (and correct) values of $y_1, y_2, \ldots$ are output from the right-most cell at the rate of one y_i per cycle. The basic principle of this design was previously proposed for circuits to implement a pattern matching processor[16] and for circuits to implement polynomial multiplication.[17-20]

Design B2—broadcast inputs, move weights, results stay. In design B2 (see Figure 4), each y_i stays at a cell to accumulate its terms, allowing efficient use of available multiplier-accumulator hardware. (Indeed, this design is described in an application booklet for the TRW multiplier-accumulator chips.[21] The weights circulate around the array of cells, and the first weight w_1 is associated with a tag bit that signals the accumulator to output and resets its contents.* In design B1 (Figure 3), the systolic path for moving y_i's may be considerably wider than that for moving w_i's in design B2 because for numerical accuracy y_i's typically carry more bits than w_i's. The use of multiplier-accumulators in design B2 may also help increase precision of the results, since extra bits can be kept in these accumulators with modest cost. Design B1, however, does have the advantage of not requiring a separate bus (or other global network), denoted by a dashed line in Figure 4, for collecting outputs from individual cells.

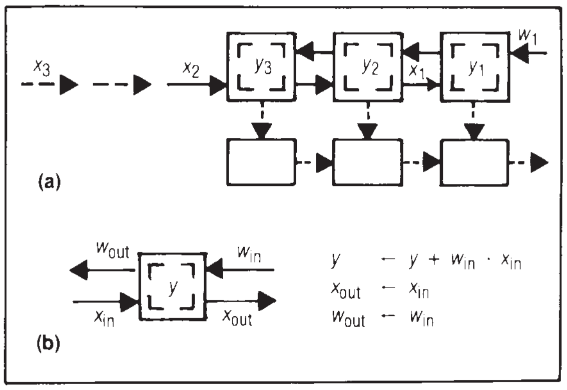
Design F—fan-in results, move inputs, weights stay. If we consider the vector of weights $(w_k, w_{k-1}, \ldots, w_1)$ as being fixed in space and input vector $(x_n, x_{n-1}, \ldots, x_1)$ as sliding over the weights in the left-to-right direction, then the convolution problem is one that computes the inner product of the weight vector and the section of input vector it overlaps. This view suggests the systolic array

*To avoid complicated pictures, control structures such as the use of tag bits to gate outputs from cells are omitted from the diagrams of this article.



x_5 x_4 x_3 x_2 x_1
w_3 w_2 w_1
ADDER
(a)
y_i's
x_{in} x_{out} w z_{out}
$z_{out} \leftarrow w \cdot x_{in}$
$x_{out} \leftarrow x_{in}$
(b)

Figure 5. Design F: systolic convolution array (a) and cell (b) where w_i's stay, x_i's move systolically, and y_i's are formed through the fan-in of results from all the cells.



w_1
x_3 x_2 x_1
y_3 y_2 y_1
(a)
w_{out} w_{in} y x_{in} x_{out}
$y \leftarrow y + w_{in} \cdot x_{in}$
$x_{out} \leftarrow x_{in}$
$w_{out} \leftarrow w_{in}$
(b)

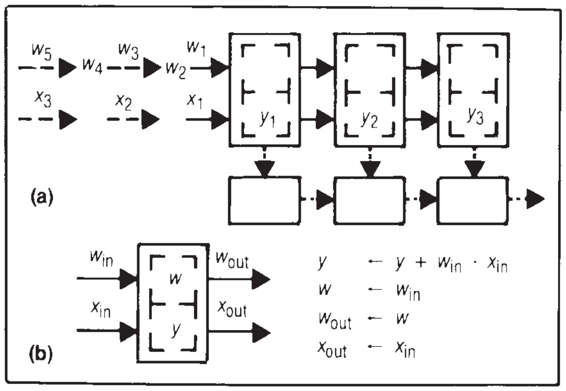
Figure 6. Design R1: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and y_i's move in opposite directions systolically.

shown in Figure 5. Weights are preloaded to the cells and stay there throughout the computation. During a cycle, all x_i's move one cell to the right, multiplications are performed at all cells simultaneously, and their results are fanned-in and summed using an adder to form a new y_i. When the number of cells, k, is large, the adder can be implemented as a pipelined adder tree to avoid large delays in each cycle. Designs of this type using unbounded fan-in have been known for quite a long time, for example, in the context of signal processing[33] and in the context of pattern matching.[43]

(Pure-) systolic convolution arrays without global data communication. Although global broadcasting or fan-in solves the I/O bottleneck problem, implementing it in a modular, expandable way presents another problem. Providing (or collecting) a data item to (or from) all the cells of a systolic array, during each cycle, requires the use of a bus or some sort of tree-like network. As the number of cells increases, wires become long for either a bus or tree structure; expanding these non-local communication paths to meet the increasing load is difficult without slowing down the system clock. This engineering difficulty of extending global networks is significant at chip, board, and higher levels of a computer system. Fortunately, as will be demonstrated below, systolic convolution arrays without global data communication do exist. Potentially, these arrays can be extended to include an arbitrarily large number of cells without encountering engineering difficulties (the problem of synchronizing a large systolic array is discussed later).

Design R1—results stay, inputs and weights move in opposite directions. In design R1 (see Figure 6) each partial result y_i stays at a cell to accumulate its terms. The x_i's and w_i's move systolically in opposite directions such that when an x meets a w at a cell, they are multiplied and the resulting product is accumulated to the y staying at that cell. To ensure that each x_i is able to meet every w_i, consecutive x_i's on the x data stream are separated by two cycle times and so are the w_i's on the w data stream.

Like design B2, design R1 can make efficient use of available multiplier-accumulator hardware; it can also use a tag bit associated with the first weight, w_1, to trigger the output and reset the accumulator contents of a cell.



w_5 w_4 w_3 w_2 w_1
x_3 x_2 x_1
y_1 y_2 y_3
(a)
w_{in} w w_{out} x_{in} y x_{out}
$y \leftarrow y + w_{in} \cdot x_{in}$
$w \leftarrow w_{in}$
$w_{out} \leftarrow w$
$x_{out} \leftarrow x_{in}$
(b)

Figure 7. Design R2: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and w_i's both move in the same direction but at different speeds.

Design R1 has the advantage that it does not require a bus, or any other global network, for collecting output from cells; a systolic output path (indicated by broken arrows in Figure 6) is sufficient. Because consecutive w_i's are well separated by two cycle times, a potential conflict—that more than one y_i may reach a single latch on the systolic output path simultaneously—cannot occur. It can also be easily checked that the y_i's will output from the systolic output path in the natural ordering $y_1, y_2, \ldots$. The basic idea of this design, including that of the systolic output path, has been used to implement a pattern matching chip.[1]

Notice that in Figure 6 only about one-half the cells are doing useful work at any time. To fully utilize the potential throughput, two independent convolution computations can be interleaved in the same systolic array, but cells in the array would have to be modified slightly to support the interleaved computation. For example, an additional accumulator would be required at each cell to hold a temporary result for the other convolution computation.

Design R2—results stay, inputs and weights move in the same direction but at different speeds. One version of design R2 is illustrated in Figure 7. In this case both the x and w data streams move from left to right systolically, but the x_i's move twice as fast as the w_i's. More precisely,

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.