A Pattern for Efficient Parallel Computation on Multicore Processors with Scalar Operand Networks

Henry Hoffmann
MIT CSAIL
hank@csail.mit.edu

Srinivas Devadas
MIT CSAIL
devadas@csail.mit.edu

Anant Agarwal
MIT CSAIL
agarwal@csail.mit.edu

ABSTRACT

Systolic arrays have long been used to develop custom hardware because they result in designs that are efficient and scalable. Many researchers have explored ways to exploit systolic designs in programmable processors; however, such efforts often result in the simulation of large systolic arrays on a general purpose platforms. While simulation can add flexibility and problem size independence, it comes at a cost of greatly reducing the efficiency of the original systolic approach. This paper presents a pattern for developing parallel programs using systolic designs to execute efficiently (without resorting to simulation) on modern multicore processors featuring scalar operand networks. This pattern provides a compromise solution that can achieve high efficiency and flexibility given appropriate hardware support. Several examples illustrate the application of this pattern to produce parallel implementations of matrix multiplication and convolution.

Introduction and Context

This paper presents a pattern for exploiting systolic designs to develop efficient parallel software for modern multicore processors. This section discusses the benefits of systolic designs, and then describes why these designs appear to be a good match for the emerging class of multicore architectures characterized by their high-performance, scalar-operand networks.

Systolic Arrays

The systolic array pattern has a long history of use in custom hardware solutions because it tends to produce designs that are highly efficient both in terms of their computational and power costs [18]. Designs based on the original concept are still in use to develop high performance embedded computing applications [22], as well as custom hardware targeting military and commercial applications in digital signal processing, communications, and multimedia [34].

Systolic array designs are characterized by a network of simple processing elements (PE). In a single clock cycle, each PE receives data from the network, performs some simple calculation on that data and injects data into the network. PEs consist of a functional unit (for performing the computation), network interfaces, and possibly a small amount of local storage in the form of registers. Data initially enters the array from peripheral memory, flows through the network, and results exit from a different edge.

As an example of a systolic array design, consider a matrix multiplication of two 2×2 matrices A and B to produce C as illustrated in Figure 1. The systolic array consists of a 2×2 array of PEs. Each PE is responsible for computing a different inner-product and thus a different element of the result matrix. Specifically, the PE in row i and column j is responsible for computing the value of c_{ij}. At cycle k, a PE reads a value a_{ik} from the network to the left, a value b_{kj} from the network above and performs a multiply-accumulate to compute $c_{ij}^{k} = c_{ij}^{k-1} + a_{ik}b_{kj}$, while forwarding a_{ik} to the right and b_{kj} down. Thus, values of A flow from left to right while values of B flow from top to bottom.

This simple example of matrix multiplication demonstrates some of the benefits of the systolic array approach. First, the array can achieve a high compute efficiency. If there are many matrix multiplications to be computed in succession (common for many signal processing applications) the entire array can be utilized such that each PE is executing a multiply-accumulate on every cycle. Furthermore, the array is efficient in the sense that all the structures required are used in every cycle, so there is no wasted area. Additionally, the design is power efficient as it 1) avoids long wires and 2) avoids using large memories on each PE. Finally, this design is scalable in that we can accommodate larger problem sizes by simply adding PEs.

Because of these advantages, many researchers have attempted to transition the systolic array concept from custom hardware designs to programmable substrates. The goal of these efforts is to create programmable processors which can take advantage of systolic designs so that their benefits are not restricted to fixed function hardware. One obvious approach to overcoming the limited flexibility of systolic arrays is to implement them on programmable hardware by *simulating* [21] or *emulating* [14] multiple processing elements on one programmable processor. This method is popular because it allows for problem-size independence and network embedding, though this comes at the cost of potentially unbounded local memory requirements. As a consequence of local memory accesses, simulated algorithms lose computational efficiency, consume significantly more energy due to

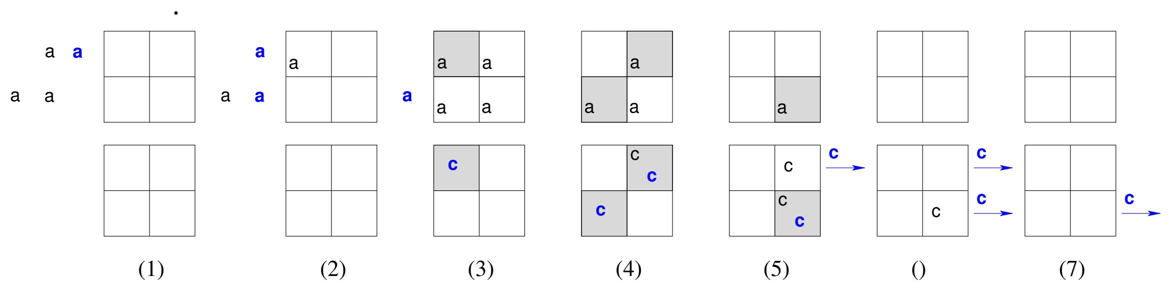

Figure 1: Seven time steps of a systolic matrix multiplication $C = A \cdot B$ for 2×2 matrices. Each box represents a processing element. Values entering, leaving, or being generated in the array are shown in bold face. Shaded boxes mark the completion of an inner product. We split the data flow of the operands and products into the top and bottom rows.

memory traffic, and can suffer from poor scalability on multicores as the memory controller becomes a bottleneck.

We would like to find a way to implement systolic array designs in software, using programmable processors, and without resorting to simulation. Such a solution would provide efficiency and scalability in combination with the flexibility to implement multiple systolic solutions for variable problem sizes using the same underlying hardware.

Multicore and Scalar-Operand Networks

Systolic designs are particularly compelling in the context of the emerging class of multicore architectures with *scalar-operand networks* [29] such as the TILE processor [33], Raw [30], Trips [23], Scale [15], Wavescalar [28], and Synchroscalar [24]. These architectures share several characteristics which make them a tempting target for systolic designs: multiple processors connected by high-bandwidth, low-latency networks. In many cases, the networks are *exposed* to software, meaning software can directly control network routing. Furthermore, these scalar-operand networks are optimized for sending small messages efficiently. In addition to their powerful networks, these architectures are characterized by a relative lack of memory bandwidth as multiple processing cores must share a memory controller.

The combination of powerful and programmable networks optimized for sending many small messages and a relative lack of memory bandwidth, makes systolic designs a good match for these architectures, as such designs favor network usage over local memory access. The scalability and efficiency of systolic designs complements the current scaling trends in modern multicore processors which tend to add cores to a chip rather than increasing the ability of individual cores. Furthermore, targeting systolic designs to general purpose multicore chips allows users to achieve efficiency when a problem can be cast as a systolic array and maintain the flexibility to solve other problems on the same architecture by using a different programming model.

This pattern for developing systolic designs for multicore provides a methodology to overcome the limitations of simulating systolic arrays and capture the efficiency and scalability of the original hardware pattern on a programmable processor. Given sufficient hardware support, the systolic software pattern achieves these goals at a cost of increased programmer effort and a lack of portability. Engineers considering the systolic software pattern should weigh their application's performance goals, their development time, and the available hardware support before making use of this pattern.

Problem

How can we use systolic designs to structure parallel code for multicore, scalar-operand networks without resorting to simulation or emulation?

Forces

- **Efficiency.** As discussed above, systolic designs for hardware are efficient in several senses. First, they achieve high computational efficiency as all the functional units are kept busy at the same time. Second, they are area efficient as they include a minimal set of structures required to solve a problem. Third, they are energy efficient because they avoid long wires and large memories. In contrast, the simulation of systolic arrays often loses this efficiency because they require (potentially unbounded) local memory access. The instructions needed to access local memory reduce computational efficiency, while the memories themselves reduce area efficiency. Accessing these memories also reduces energy efficiency compared to memory access. For example, a recent study showed that in 90nm technology, the cost of sending one 32-bit word to a neighbor is approximately 5 pJ while retrieving one 32-bit word from a 32 KB data cache costs approximately 50 pJ [25].

- **Scalability.** Systolic designs for hardware are scalable in the sense that additional PEs can be added to a design without sacrificing efficiency. In contrast, simulation of a systolic array in software can lose this scalability due to local memory accesses. As the amount of local memory grows the energy efficiency decreases. Alternatively, local memory can be backed up using additional external memory (such as a DRAM); however, the resulting DRAM accesses can become a bottleneck that limits scalability. This is especially true when simulating systolic arrays on modern multicore processors in which multiple cores to compete for DRAM bandwidth as described in [12].

- **Flexibility.** Simulation of a systolic design on programmable hardware is flexible. The structures and abstractions required to simulate a variable number of PEs on a single processor allow multiple systolic designs to be simulated using the same underlying hardware. Additionally, such programmable hardware can also execute code that does not lend itself to a systolic design. In contrast, systolic hardware implementations are inflexible as they are designed to solve a single problem and cannot be re-purposed.

- **Problem Size Independence.** The simulation of systolic designs also allows problem size independence. Modifying the design to work on larger problem sizes simply requires allowing a single physical processor to simulate a larger number of systolic PEs. Systolic designs for hardware are built for a fixed problem size and require adding additional physical PEs to accommodate the additional input sizes.

We want to find a compromise solution, providing most of the efficiency and scalability of the systolic approach while the maintaining flexibility and problem size independence of a more general purpose approach suitable for multicore implementation.

Solution

We design parallel programs using on two key strategies: 1) partition the application into processors[1] which access memory and those which perform computation (referred to as memory and compute processors respectively) and 2) make the number of memory processors (M) negligible compared to the number of compute processors (P). While memory processors are allowed to access memory, compute processors are restricted to operating on data from the network or stored in local registers. Thus, the compute processors form a programmable systolic array and the memory processors provide a mechanism for feeding the array with data. This strategy is motivated by the fact that cores busy accessing memory are not contributing useful computation (e.g. flops), yet tend to consume a relatively large amount of energy.

We aim for the condition where $M = o(P)$. Informally, if this condition holds the number of memory processors becomes insignificant relative to the number of compute processors as we increase the total number of processors. This relationship between M and P is a necessary condition to amortize the lack of useful computation performed by the memory processors and the power invested in memory accesses. For example, suppose we implement an algorithm on a 2D mesh of $P = R^2$ compute processors. If we can arrange the memory processors such that their number M becomes negligible compared to P when increasing R, the resulting algorithm meets this condition. Thus, for a design with $P = \Theta(R^2)$, we may choose M to be $O(\lg R)$, or $O(\sqrt{R})$, or $M = O(R)$, or $M = O(R \lg R)$, for example. In contrast, an algorithm designed to simulate a large systolic array time-multiplexing one physical processor among multiple virtual processors (as described in [14]) would require memory to buffer the state of each simulated processor. Thus each processor would have to be a memory processor and $M = \Omega(R^2)$ our condition would not hold.

[1]This work assumes that each processor is running a single thread or process.

To apply the systolic software pattern, we start with a systolic algorithm designed for a hardware implementation (in which the required number of cells is proportional to the size of the problem being solved). We assume that the number of processors required in a hardware implementation of this algorithm is Q and that the total number of processors R on the target architecture is much less than Q ($Q >> R$). We convert the systolic algorithm to one which can be executed efficiently on our architecture by applying the following two step process based on the design of stream algorithms [11].

Partitioning: We start by partitioning the problem into smaller, independent subproblems. Each of the subproblems, as well as the composition of their results, must be suitable for parallelization by means of a smaller systolic algorithm such that the compute processors access data in registers and on the network only. For simple applications, the partitioning can be obvious immediately. For applications with more complicated data dependencies, we find that recursive formulations and partitioning methods like those developed for out-of-core algorithms [32] can be helpful. We continue to recursively apply partitionings until each sub-problem is sized such that it can execute systolically, without resorting to simulation or emulation, on the available compute processors.

Decoupling: Our goal is to move the memory accesses off the critical path. To this end, we have to decouple the computation such that the memory accesses occur on the memory processors and compute operations on the compute processors (a design principle shared by the the decoupled-access execute architecture [27]) . For a systolic problem, the memory processors feed the input streams into the compute processors, and the decoupling procedure is almost trivial. However, the composition of several subproblems requires careful planning of the flow of intermediate data streams, such that the output streams of one systolic phase can become input streams of a subsequent phase without copying streams across memory processors. Occasionally, it is beneficial to relax the strict dedication of memory processors to memory accesses, and compute portions of the composition of the subproblems, such as reductions, on the memory processors themselves.

In practice this is often an iterative process; sometimes promising partitionings are unable to be decoupled.

Examples

This section presents two examples applying the systolic software pattern to first a matrix multiplication and second a convolution. These examples are taken from formulations of stream algorithms as described in [11] and were used to develop efficient software implementations exploiting the scalar operand network on the Raw processor [31].

Matrix Multiplication

As our first example, we consider a dense matrix multiplication. Given two $N \times N$ matrices A and B, we wish to compute the $N \times N$ matrix $C = AB$. We compute element

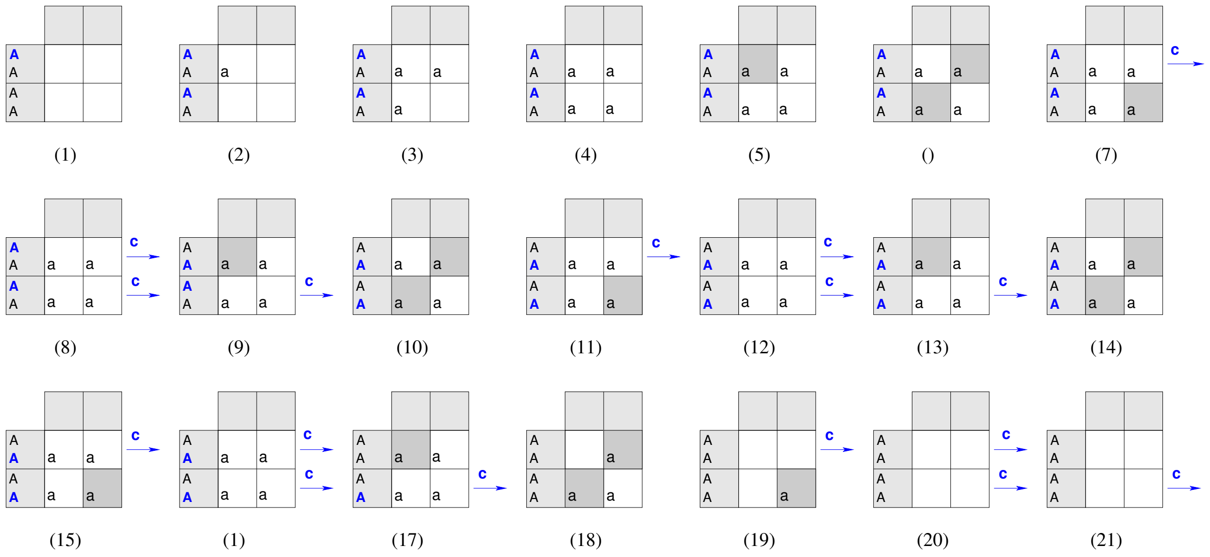

Figure 2: Data flow of a matrix multiplication $C = A \cdot B$ for 4×4 matrices on 2×2 compute processors. Shaded boxes on the periphery mark memory processors, and indicate the completion of an inner-product otherwise.

c_{ij} in row i and column j of product matrix C as the inner product of row i of A and column j of B:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}, \qquad (1)$$

where $1 \le i, j \le N$.

Partitioning

We use a block-recursive partitioning for the matrix multiplication. We recurse along the rows of A and the columns of B:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \end{pmatrix}. \qquad (2)$$

For each of the matrices C_{ij} we have $C_{ij} = A_{i1}B_{1j}$, where A_{i1} is an $N/2 \times N$ matrix and B_{1j} an $N \times N/2$ matrix. Thus, the matrix multiplication can be partitioned into a homogeneous set of subproblems.

Decoupling

We begin by observing that each product element c_{ij} can be computed independently of all others by means of Equation 1. In addition, Equation 2 allows us to stream entire rows of A and entire columns of B through the compute processors. Furthermore, we partition a problem of size $N \times N$ until the C_{ij} are of size $R \times R$ and fit into our array of compute processors. We implement the resulting subproblems as systolic matrix multiplications, illustrated in Figure 1 for $N = R = 2$. Rows of A flow from the left to the right, and columns of B from the top to the bottom of the array.

For $N > R$, the compute processor in row r and column s computes the product elements c_{ij} for all $i \bmod R = r$ and $j \bmod R = s$. To supply the compute processors with the proper data streams, we use R memory processors to store the rows of A and R additional memory processors to store the columns of B. Thus, for the matrix multiplication, we use $P = R^2$ compute processors and $M = 2R$ memory processors. Figure 2 illustrates the data flow of a decoupled systolic matrix multiplication for $N = 4$ and $R = 2$. Note how the memory processors on the periphery determine the schedule of the computations by streaming four combinations of rows of A and columns of B into the compute processors. First, we compute C_{11} by streaming $\{A(1,:),\ A(2,:)\}$ and $\{B(:,1),\ B(:,2)\}$ through the array. Second, we stream $\{A(1,:),\ A(2,:)\}$ against $\{B(:,3),\ B(:,4)\}$, third, $\{A(3,:),\ A(4,:)\}$ against $\{B(:,1),\ B(:,2)\}$, and finally $\{A(3,:),\ A(4,:)\}$ against $\{B(:,3),\ B(:,4)\}$. As a result, we compute C_{11}, C_{12}, C_{21}, and C_{22} in that order.

If product matrix C cannot be streamed into a neighboring array of consuming compute processors or off the chip altogether, but shall be stored in memory processors, we may have to invest another R memory processors for a total of $M = 3R$. In any case, we have $P = \Theta(R^2)$ and $M = \Theta(R)$, and hence $M = o(P)$.

Note that we could use a similar organization to compute a matrix-vector product Ax, where A is an $N \times N$ matrix and x an $N \times 1$ vector. However, using only one column of $R \times 1$ compute processors requires $M = R + 1$ memory processors. Since $M \neq o(P)$, this organization is not efficient. However, there exists a different design that is efficient by storing matrix A and vector x on one memory processor and by distributing the inner products across a linear array of compute processors.

Convolution

The convolution of sequence $[a]$ of length $N_{samples}$ with sequence $[w]$ of length N_{taps} produces an output sequence $[b]$ of length $N_{samples} + N_{taps} - 1$. Without loss of generality, we assume that $N_{samples} \ge N_{taps}$. Element k of $[b]$ is given by

$$b_k = \sum_{i+j=k+1} a_i \cdot w_j \qquad (3)$$

where

$$\begin{aligned} 1 \leq \quad & k \quad \leq N_{samples} + N_{taps} - 1 \\ 1 \leq \quad & i \quad \leq N_{samples} \\ 1 \leq \quad & j \quad \leq N_{taps}. \end{aligned}$$

Partitioning

We partition the convolution into N_{taps}/R subproblems by partitioning the sum in Equation 3 as follows:

$$b_k \quad = \quad \sum_{l=1}^{N_{taps}/R} \quad \sum_{i+j=k+1} a_i \cdot w_j \qquad (4)$$

where

$$\begin{aligned} 1 \leq \quad & k \quad \leq N_{samples} + R - 1 \\ 1 \leq \quad & i \quad \leq N_{samples} \\ (l-1)R + 1 \leq \quad & j \quad \leq lR + 1. \end{aligned}$$

This partitioning expresses the convolution of $[a]$ and $[w]$ as the sum of convolutions of $[a]$ with N_{taps}/R weight sequences $[w]_j$. Intuitively, we partition weight sequence $[w]$ into chunks of length R, compute the partial convolutions, and exploit the associativity of the addition to form the sum of the partial convolutions when convenient.

Decoupling

We use the systolic design of Figure 3 to implement a convolution with $N_{taps} = R$. This design is independent of the length $N_{samples}$ of sequence $[a]$. For the example in Figure 3 we have chosen $N_{taps} = R = 4$ and $N_{samples} = 5$. Both sequence $[a]$ and weight sequence $[w]$ enter the array from the left, and output sequence $[b]$ leaves the array on the right. Computation tile p_i is responsible for storing element w_i of the weight sequence. Thus, the stream of elements w_i folds over on the way from left to right through the array. In contrast, sequence $[a]$ streams from left to right without folding over. During each time step, the compute processors multiply their local value w_i with the element of a_j arriving from the left, add the product to an intermediate value of b_k that is also received from the left, and send the new intermediate value to the right. The elements of $[b]$ leave the array on the right.

We illustrate the data movement in Figure 3 by discussing the computation of $b_4 = a_4w_1 + a_3w_2 + a_2w_3 + a_1w_4$. We begin with time step 5 in Figure 3 when element a_4 enters tile p_1 on the left. Element w_1 is already resident. Tile p_1 computes the intermediate value $b_4^1 = a_4 \cdot w_1$, and sends it to tile p_2. At time step 6, p_2 receives a_3 and b_4^1 from tile p_1 on the left. With weight w_2 already resident, tile p_2 computes intermediate value $b_4^2 = b_4^1 + a_3 \cdot w_2$. In time step 7, values b_4^2, a_2, and w_3 are available for use by tile p_3. It computes and sends intermediate value $b_4^3 = b_4^2 + a_2 \cdot w_3$ toward tile p_4. At time step 8, p_4 receives b_4^3, a_1, and w_4 from p_3, and computes $b_4 = b_4^3 + a_1 \cdot w_4$. At time step 9, b_4 exits the computation array.

We use the partitioning of Equation 4 to reduce a convolution with a weight sequence of length N_{taps} into N_{taps}/R systolic convolutions that match network size R of a linear array of compute processors. In addition, we employ one memory tile on the left of the array to buffer sequences $[a]$ and $[w]$, and another memory tile on the right of the array to store intermediate values of the computation as well as to compute the sum of the subproblems. Figure 4 illustrates the computation of a convolution on a linear processor array. Our decoupled systolic convolution requires $P = R$ compute processors and $M = 2$ memory processors. We observe that $M = o(P)$ and, therefore, our convolution is efficient.

Performance

This section shows the performance and scalability benefits that are possible using the systolic software approach. Results in this section use the Raw simulator. Raw is one example of a multicore processor featuring a scalar operand network[2], and its simulator allows simulations of Raw fabrics of up to 1024 cores.

We explore the scalability of software designed with this approach by reconfiguring the Raw simulator to support larger numbers of cores and measuring the compute efficiency of matrix multiplication and the QR factorization as a function of the number of cores. For this experiment we fix the problem size so that each matrix operates on square matrices of size 1024×1024 and we vary the total number of processors. We measure the compute efficiency of these algorithms as the fraction of the peak flop rate that they achieve.

P	Matrix Multiply	QR Factorization
16	0.95	0.92
64	0.96	0.88
256	0.96	0.80
1024	0.95	0.68

Table 1: Compute efficiency of two algorithms implemented on Raw using the systolic software pattern. P is the number of processors used. Each problem uses 1024×1024 matrices. Compute efficiency is measured as the fraction of the peak flop rate achieved by that implementation.

As shown in Table 1, the systolic software pattern can lead to highly scalable parallel implementations. The matrix multiply achieves nearly perfect scaling moving from 16 to 1024 processing cores. The QR factorization also achieves good scaling and is able to achieve 68% of the peak flop rate on 1024 cores.

These results show that the use of the systolic software pattern can result in high-performance scalable implementations. Users will have to weigh these benefits against the tradeoffs of programming difficulty and lack of portability as described in the forces section.

Forces Resolved

- **Computational efficiency.** As demonstrated above, the use of this pattern results in code that is computationally efficient when executed on a multicore with a scalar operand network. This efficiency comes from the structure of the computation. Loads and stores are removed from the critical path and instead processors execute only operations defined by the computation and possibly branch instructions. By eliminating memory accesses on the critical path, the program executes fewer dynamic instructions; more importantly,

[2]In fact, raw has four.

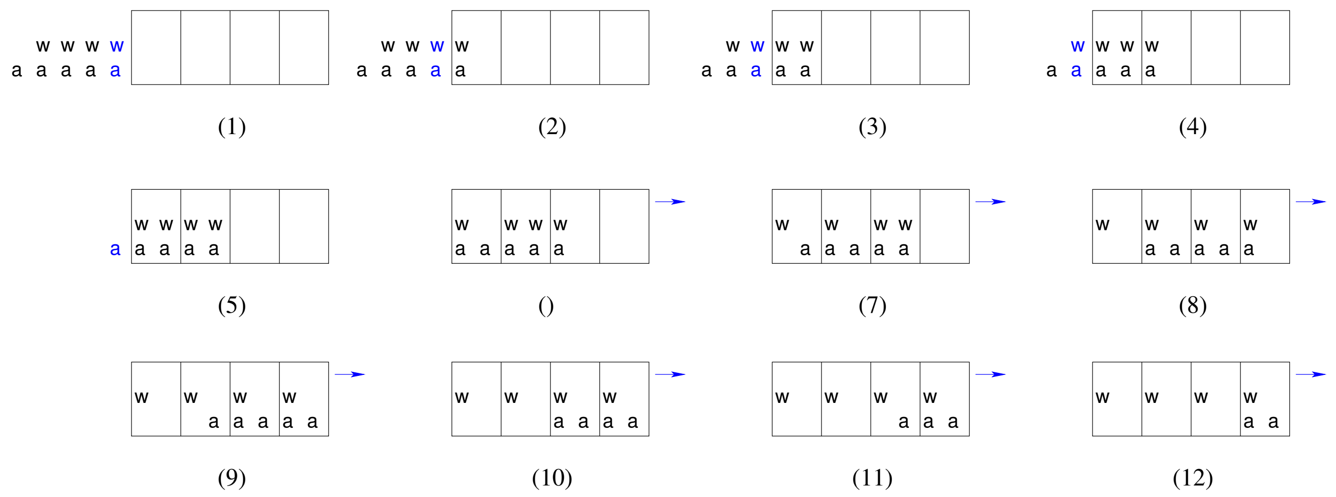
Figure 3: Systolic convolution of a sequence of input values a_i of length $N_{samples} = 5$ with $N_{taps} = 4$ weights w_j. Both the weights and input sequence are fed into the linear array of $R = 4$ compute processors. Intermediate results are shown above the corresponding processors. Value b_k^i represents an intermediate value of b_k after the first i products have been computed according to Equation 3.

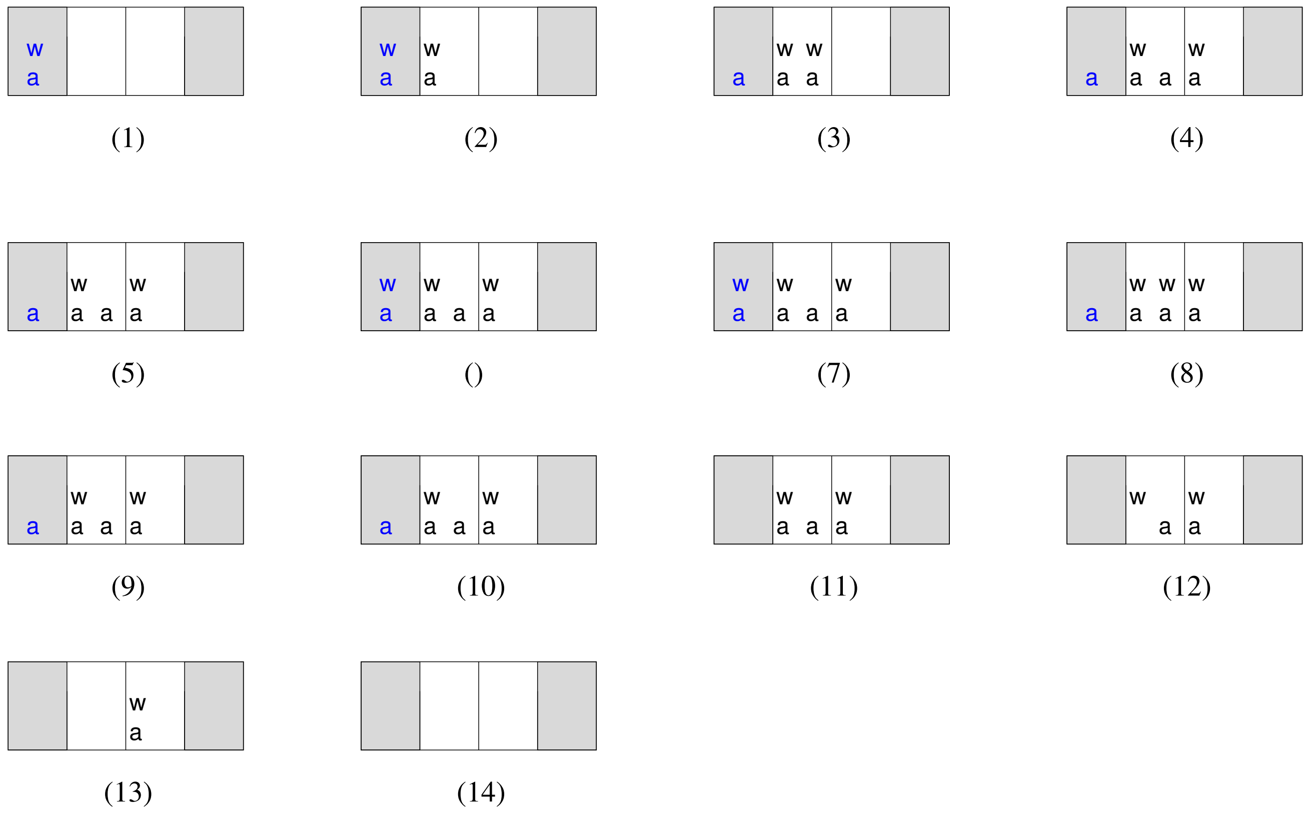
Figure 4: Systolic software convolution of an input sequence of length $N_{samples} = 5$ with $N_{taps} = 4$ weights on a linear array of $R = N_{taps}/2 = 2$ compute processors and $M = 2$ memory processors. Value $b_k^{l,i}$ represents the computation of b_k when the outer summation of Equation 4 has been executed l times and the inner summation has been executed i times. Note that the memory processor on the right performs an addition to accumulate the results of the partial convolutions.

it also removes cache misses from the critical path. As shown in the example, the systolic software pattern can provide as much as 5× the speed of a design which uses loads and stores.

- **Power efficiency.** Use of the systolic software pattern produces implementations which favor near-neighbor communication over the access of their own local caches. Sending data to a neighbor rather than storing it locally can save considerable energy on modern multicore processors [25]. Clearly programs which favor communication over local storage can greatly reduce energy consumption.

- **Scalability.** The systolic software pattern captures the scalability of the original systolic array pattern for hardware. Using the systolic software pattern all network communication is limited to a processor's nearest neighbors so there are no bottlenecks in the network. Furthermore, memory access is removed from the critical path of computation. Following this pattern the number of processors that access memory (M) is asymptotically smaller than the number of processors which perform the computation (P); $M = o(P)$. Thus, memory is also prevented from becoming a bottleneck. As demonstrated the matrix multiplication based on this pattern achieves 95% floating-point efficiency on an 1024 core multicore, while systolic software implementation of a QR factorization achieves a floating point efficiency of 68% on the same 1024 core multicore [10].

- **Flexibility and problem size independence.** The methodology provided with this pattern allows the flexibility and problem size independence associated with the simulation of systolic arrays without resorting to local memory access.

Limitations

- **Programming effort and maintainability.** Perhaps the biggest limitation affecting the use of this pattern is the increased effort of developing the program. Several factors contribute to this difficulty. First, achieving the above benefits requires understanding some low-level details of the underlying hardware. Second, making use of this pattern requires programmers to schedule communication and computation on a word-level granularity. Such fine grain scheduling is often difficult for engineers who are used to thinking in terms of phases of bulk processing interleaved with bulk communication. Finally, developing programs with this pattern tends to result in *steep slope* development where the program does not work at all until all scheduling issues are fixed when it suddenly works and runs extremely efficiently. This steep slope tends to frustrate programmers more than other patterns which allow for incremental improvement towards a performance goal.

- **Hardware support and portability.** Although a programming model could be implemented allowing the systolic software pattern to be applied on any hardware platform, the benefits are only achievable with proper hardware support. Because proper hardware support can provide such benefits in efficiency, power, and scalability, several hardware platforms have been designed explicitly to support systolic software development such as the Wavefront Array Processor [19], PSC [3], Warp [1], iWarp [8], Saxpy's Matrix-1 [4], and the CM2 [9].

  Looking forward, compiler research on instruction level parallelism (ILP) [26, 20] is influencing the design of multicore processors. Such compilers also require hardware support for fine-grain scheduling of communication and computation and benefit from the same network architecture needed to take full advantage of the systolic software pattern. Processors such as the TILE processor [33], Raw [30], Trips [23], Scale [15], Wavescalar [28], and Synchroscalar [24] all provide support for ILP using programmable processing elements with small amounts of memory communicating via on-chip scalar operand networks. Given this trend, it is possible that many future multicore processors will support such networks and thus provide good targets for the systolic software pattern.

- **Applicability.** The benefits of the systolic software pattern are, of course, limited to those applications for which systolic formulations can be found. Systolic algorithms currently exist for a wide range of dense matrix problems commonly found in scientific computing and digital signal processing [18, 2] as well as some sorting and graph problems [21]. The systolic pattern is not a good match for highly irregular applications where communication patterns cannot be predicted in advance.

Known Uses

H.T. Kung advocates *systolic communication* for the Warp and iWarp machines, and reports on software implementations for these architectures show various combinations of a systolic software approach similar to that advocated in this pattern and the less-efficient, but easier to implement, simulation of systolic arrays[1, 8, 16, 17, 6]. Additionally, Gross and Lam describe a compiler for Warp that exploits the systolic software pattern when possible [7].

Several programmable systolic architectures are designed to run software exploiting the systolic software pattern [19, 3, 4].

This pattern was used to generate highly efficient software implementations of several dense linear algebra problems for the Raw architecture as described in [31].

The StreamIt compiler produced a back-end for Raw, which exploited the systolic software pattern when possible [5].

Related Patterns

The systolic software pattern may be useful for many applications based on OPL's Dense Linear Algebra pattern and possibly some applications or kernels which fall under the Graph Algorithms pattern [13].

Systolic arrays themselves exploit extremely fine grain pipelines and data parallelism, so the systolic software pattern could be viewed as a concurrent execution pattern combining OPL's Pipeline and Data Parallel algorithm strategy patterns [13].

To exploit the systolic software pattern a program must have loop level parallelism so a systolic implementation may be worth exploring for applications that use OPL's Loop Parallelism pattern.

Acknowledgements

Thanks to the attendees of ParaPLoP 2010, and especially Ralph Johnson, who provided many helpful comments on earlier versions of this paper.

1. REFERENCES

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb. Warp Architecture and Implementation. In *13th Annual Symposium on Computer Architecture*, pages 346–356, 1986.

[2] R. P. Brent, F. T. Luk, and C. F. V. Loan. Computation of the Singular Value Decomposition Using Mesh-Connected Processors. *Journal of VLSI and Computer Systems*, 1(3):242–260, 1985.

[3] A. L. Fisher, H. T. Kung, L. M. Monier, and Y. Dohi. Architecture of the PSC—A Programmable Systolic Chip. In *10th International Symposium on Computer Architecture*, pages 48–53. IEEE Computer Society Press, 1983.

[4] D. E. Foulser and R. Schreiber. The Saxpy Matrix-1: A General-Purpose Systolic Computer. *IEEE Computer*, 20(7):35–43, July 1987.

[5] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffmann, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[6] T. Gross, S. Hinrichs, D. R. O'Hallaron, T. Stricker, and A. Hasegawa. Communication Styles for Parallel Systems. *IEEE Computer*, 27(12):34–44, Dec. 1994.

[7] T. Gross and M. S. Lam. Compilation for a High-performance Systolic Array. In *SIGPLAN 86 Symposium on Compiler Construction, ACM SIGPLAN*, pages 27–38, 1986.

[8] T. Gross and D. R. O'Hallaron. *iWARP: Anatomy of a Parallel Computing System*. MIT Press, Cambridge, MA, 1998.

[9] D. W. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[10] H. Hoffmann and A. Agarwal. A quantitative analysis of stream algorithms on raw compute fabrics. In *Third Annual Boston Area Architecture Workshop.*

[11] H. Hoffmann, V. Strumpen, and A. Agarwal. Stream Algorithms and Architecture. Technical Memo MIT-LCS-TM-636, Laboratory for Computer Science, Massachusetts Institute of Technology, Mar. 2003.

[12] E. Ipek, O. Mutlu, J. F. MartŠnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch.*, 2008.

[13] K. Keutzer and T. Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software.

[14] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-Preserving Emulations of Fixed-Connection Networks. *Journal of the ACM*, 44(1):104–147, Jan. 1997.

[15] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *31st International Symposium on Computer Architecture*, München, Germany, June 2004. (publication pending).

[16] H. T. Kung. Systolic Communication. In *International Conference on Systolic Arrays*, pages 695–703, San Diego, CA, May 1988.

[17] H. T. Kung. Warp Experience: We Can Map Computations Onto a Parallel Computer Efficiently. In *2nd International Conference on Supercomputing*, pages 668–675. ACM Press, 1988.

[18] H. T. Kung and C. E. Leiserson. Algorithms for VLSI Processor Arrays. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 8.3, pages 271–292. Addison-Wesley, 1980.

[19] S.-Y. Kung, R. J. Gal-Ezer, and K. S. Arun. Wavefront Array Processor: Architecture, Language and Applications. In *Conference of Advanced Research in VLSI*, pages 4–19, M.I.T., Jan. 1982.

[20] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, Oct. 1998.

[21] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[22] D. R. Martinez, R. A. Bond, and M. M. Vai. *High Performance Embedded Computing Handbook*. CRC Press, Boca Raton, 2008.

[23] R. Nagarajan, K. Sankaralingam, D. C. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *34th Annual International Symposium on Microarchitecture*, pages 40–51, Dec. 2001.

[24] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. Jones IV, D. Copsey, D. Keen, V. Akella, and F. T. Chong. Synchroscalar: A Multiple Clock Domain Power-Aware Tile-Based Embedded Processor. In *31st International Symposium on Computer Architecture*, München, Germany, June 2004. (publication pending).

[25] J. Psota, J. Eastep, J. Miller, T. Konstantakopoulos, M. Watts, M. Beals, J. Michel, K. Kimerling, and A. Agarwal. Atac: On-chip optical networks formulticore processors. In *Fifth Annual Boston Area Architecture Workshop*, January.

[26] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective. *The Journal of Supercomputing*, 7(1):9–50, May 1993.

[27] J. E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.

[28] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *36th International Symposium on Microarchitecture*, pages 291–302, San Diego, CA, Dec. 2003. IEEE Computer Society.

[29] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of HPCA 2003*, February 2003. Also http://www.cag.lcs.mit.edu/raw/.

[30] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–36, March/April 2002.

[31] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture*, June 2004.

[32] S. Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

[33] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[34] R. F. Woods, J. V. Mccanny, and J. G. Mcwhirter. From bit level systolic arrays to hdtv processor chips. *J. Signal Process. Syst.*, 53(1-2):35–49, 2008.