Abstract—Full use of the parallel computation capabilities of present and expected CPUs and GPUs require use of vector extensions. Yet many actors in data flow systems for digital signal processing have internal state (or, equivalently, an edge that loops from the actor back to itself) that impose serial dependencies between actor invocations that make vectorizing across actor invocations impossible. Ideally, issues of inter-thread coordination required by serial data dependencies should be handled by code written by parallel programming experts that is separate from code specifying signal processing operations. The purpose of this paper is to present one approach for so doing in the case of actors that maintain state. We propose a methodology for using the parallel scan (also known as prefix sum) pattern to create algorithms for multiple simultaneous invocations of such an actor that results in vectorizable code. Two examples of applying this methodology are given: (1) infinite impulse response filters and (2) finite state machines. The correctness and performance of the resulting IIR filters are studied.

Index Terms—digital signal processing, data flow computing, vector processors, parallel algorithms

I. INTRODUCTION

Processors available for embedded systems increasing provide two forms of data parallelism. One form is the use of a graphics processor unit (GPU) for compute purposes [1]. The other form of data parallelism is the increase in the bit width of vectors of integer or floating point numbers that can be operated on by a single instruction in a single instruction multiple data (SIMD) fashion. For example, Intel and AMD have added the Advanced Vector Extensions (AVX) [2][3] capability to their processors. Some ARM designs include NEON, a similar vector SIMD instruction set extension. Use of such vector extensions has been found to increased speed of numerically-intensive tasks reasonably similar to signal processing while reducing power consumption per computation. [4] Multicore CPUs with such vector extensions have been found (by workers examining numerical simulation workloads not too dissimilar from DSP workloads) to achieve approximately one third of the price-performance (i.e., gigaflops per dollar) and power-performance (i.e., gigaflops per watt) of a GPU external to the CPU [5], without the real estate requirements of an external GPU. Both GPUs and vector extensions provide fine grained parallelism in the form of the ability to fetch or store multiple data items from or to evenly strided memory locations and perform several instances of the same mathematical operation on different data with one machine instruction.

There are three options for programming vector extensions. One is assembly language, which is so time-consuming that it is rarely practical. The second is to code in standard imperative languages (e.g., C, C++) and compile using vectorizing compilers. Such compilers attempt to locate parallelism in inner loops that has no or few enough and the right sort of data dependencies between loop iterations that vector instructions can be generated that compute the results of several adjacent loop iterations at once. If such an attempt is successful, we say that the loop has been vectorized. However, different compilers and different versions of the same compiler differ wildly in their ability successfully to vectorize the same loop. [6]

The third option is to write the most compute-intensive loops a kernel language such as OpenCL [7] or CUDA [8]. In such languages, parallelism is made explicit by separating the specification of computations that may proceed in parallel (called “work items” in OpenCL), from the specification of both the size and location of the data on which to compute. A compiler for a kernel language can produce code that uses vector extensions to execute several work items in parallel. Kernel languages have therefore been found to take better advantage both of vector extensions and opportunities for multithreading than do vectorizing compilers [9]. Furthermore, the only way to program GPUs is using either OpenCL or CUDA.

Now let’s consider using the parallelism made usable by implementing data flow actors in a kernel language. Suppose an actor maintains no internal state, that is, never writes to any variables that may be read during a later firing of the actor. Then a kernel language implementation that effectively obtains parallelism is straightforward. First, a schedule is chosen where the actor is to be fired some number $N$ times in a row. That portion of the schedule is then implemented by invoking a kernel implementing the actor functionality where the work required is vectorized. However, different compilers and different versions of the same compiler differ wildly in their ability successfully to vectorize the same loop. [6]

This straightforward strategy breaks down when the actor has an edge that is a self-loop, that is, an edge that begins and ends at that edge. Suppose that data placed on the edge are specified to arrive at the next execution of the actor. Then this self-loop is equivalent to the actor having one or more internal variables, that is, internal state, maintained within the actor. Indeed, some dataflow formalisms provide the appearance of
having internal state as a convenience for programmers while implementing it as a self-loop. Even though many data flow formalisms do not permit internal state, below we write about actors with internal state with the understanding that they can be specified in those formalisms with the aid of self-loops.

Having internal state means that past values of the actor’s inputs may influence all future outputs. Two simple examples of this (which will be referred to repeatedly below) are an actor that represents an infinite impulse response (IIR) filter and an actor that implements a finite state machine (FSM). In the case of an IIR filter, the state is some fixed number of previous filter output values. In the case of an FSM, it is the current state. One may not simply invoke a kernel implementing the actor on \( N \) data items because the operation of the actor operating on the \( i \)th data item depends on the state after the completion of the actor’s computation on the \((i-1)st\) data item.

Kernel languages include coordination mechanisms such as barriers and fences to permit information to be passed between work items via shared memory. However, correctly programming using such mechanisms is time consuming and error prone. The clarity of code is also reduced because the desired signal processing functionality is mixed in with these synchronization considerations. Ideally, issues of inter-workgroup coordination required by serial data dependencies should be handled by code written by parallel programming experts that is separate from code specifying signal processing operations. The purpose of this paper is to present one approach for so doing in the case of actors that maintain state.

We do not consider the problem of arriving at a schedule that permits the large number of consecutive actor firings that allow such vectorized code to run effectively. Such scheduling is considered in [10].

II. PROPOSED METHODOLOGY

The approach proposed here is based on identifying the transformation that an actor performs on its state variables as a function of each input. If any two such transformations can be composed with reasonable efficiency, then there is a parallel method for composing a large number of transformations with little additional overhead.

To formalize this notion, let \( x_1, x_2, \ldots \) be the scalar- or vector-valued inputs to the actor at the first, second, and so on, on invocation of the actor. Let \( s_0 \) be the initial values of the variables local to the actor that represent its internal state. Let \( s_i \) represent the values of those variables after the actor reads and processes inputs \( x_i \). With out loss of generality, the outputs of the actor are either (1) a subset of the state variables or (2) may be computed from the state variables simply and quickly. The mapping performed by the actor can then be written as a function \( f \) where \( s_i = f(s_{i-1}; x_i) \). Or, once input \( x_i \) is available, we can write \( s_i = f_i(s_{i-1}) \) where \( f_i(s) = f(s, x_i) \). Now suppose that the actor is scheduled to run \( N \) times successively and the first invocation of this block of \( N \) is the \( k \)th one during the program execution. Then the successive values of the state variables are \( f_k(s_{k-1}) \), then \( (f_{k+1} \circ f_k)(s_{k-1}) \), then \( (f_{k+2} \circ f_{k+1} \circ f_k)(s_{k-1}) \), and so on, where \( \circ \) represents functional composition. The final value of the state variables is \((f_{k+N-1} \circ \ldots \circ f_k)(s_{k-1})\).

Because the actor outputs are among the state variables, all of these functional compositions must be computed. However, since functional composition is an associative operation, the parallel prefix [11], also called parallel scan [12], pattern can be applied to yield a parallel algorithm for computing all of the required functional compositions at once. The details of the scan algorithm are beyond the scope of this article: they may be found in the references cited in this paragraph. What is important about scan for the present purposes is:

- Given objects \( a_1, a_2, \ldots, a_n \) from a set \( C \) and an associative binary operation \( \oplus \) on \( C \), scan computes \( a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \ldots, a_1 \oplus \ldots \oplus a_n \). (Some implementations also require a neutral element object of the same type as the \( a_i \’ s \), such that \( e \oplus a = a \) for all \( a \in C \).)
- When there are sufficient processors or parallel lanes in vector mode instructions, the parallel execution of the scan requires \( O(\log n) \) time using \( O(n) \) calls to the \( \oplus \) function.
- There are a number of implementations of scan available in kernel languages [13]-[16]. These implementations encapsulate the barriers, fences, and multiple kernel invocations required in a correct and efficient implementation of scan and provide an interface through which the using programmer provides the \( \oplus \) function. Generally the \( \oplus \) function is relatively straightforward to write because it must not have dependencies on variables other than its arguments and so can not use barriers, fences, or other inter-thread synchronization or communication.

The basic idea of the remainder of this paper is to use the functional composition operator \( \circ \) as \( \oplus \) and apply scan to compute the chains of functional compositions \( f_{k+1} \circ f_k, f_{k+2} \circ f_{k+1} \circ f_k \) and so on. This results in following methodology for producing algorithms suitable for implementing actors for processors with vector features using kernel languages.

1) Identify a representation for the state space \( s \).
2) Identify a representation for the state transformations \( f_i \), including how to create \( f_i \) given actor input \( x_i \). The representation must also be suitable for representing all possible compositions of the state transformations.
3) Create an efficient algorithm for composing the state transformations.

The reason that this is a methodology and not an algorithm is that some creativity is often required in order to arrive at a suitable representation and composition algorithm. Also there is no guarantee that a parallel algorithm that has net speedup will result.

Now we present two examples of applying this methodology, namely scan-based parallel actors for IIR filters and finite state machines.
III. Example 1: IIR filters

Consider an IIR filter whose $i$th output given inputs $x[1], x[2], \ldots, x[i]$ is

$$y[i] = c_1 y[i - 1] + \cdots + c_Q y Q x[i - Q] + c_{Q+1} x[i] + c_{Q+2} x[i - 1] + \cdots + c_{Q+P} x[i - P]$$

(1)

The state required to compute the succeeding output $y[i]$ contains the previous $Q$ outputs and $P - 1$ previous inputs, or as a vector, $s_{i-1} = (y[i - 1], y[i - 2], \ldots, y[i - Q], x[i - 1], x[i - 2], \ldots, x[i - P])^T$. (The reason for the constant, final element 1 is so that an affine operation can be represented in a matrix.) The transformation $f_i$ can be represented by a matrix $M(x_i)$, a function of $x_i$.

$$M(x_i) = \begin{bmatrix} c_0 & \cdots & c_Q & c_{Q+2} & \cdots & c_{Q+P-1} & c_{Q+P} & c_{Q+1} x_i \\ 1 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 1 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 1 \\ 1 & \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots & 1 \end{bmatrix}$$

(2)

The $r$th row of $M(x_i)$ tells how to compute the value of the $r$th state variable after the $i$th firing of the IIR filter. It can be verified by substitution that this matrix does map the state vector at step $i - 1$ to that at step $i$, i.e., $s_i = M(x_i) s_{i-1}$. Since the composition of the transformations represented by two matrices is their matrix product, composition of the transformations $f$ is performed by multiplication of the matrix representations $M$. That is, for any $m > 0$, $f_k \circ \cdots \circ f_1 = M(x_{k+m}) M(x_{k+m-1}) \cdots M(x_k)$.

This yields the following algorithm for an IIR filter actor.

At application start time, the state vector $s$ of the actor is initialized. Suppose that the actor is invoked $N$ successive times on input data $x_k, \ldots, x_{k+N-1}$. Then $N$ outputs $y_k, \ldots, y_{k+N-1}$ and the updated value of $s$ are computed thus:

1) Compute $M(x_k), \ldots, M(x_{k+N-1})$ in parallel. (There are no data dependencies among these computations.)

2) Apply the scan operation to the matrices, where the binary operator is matrix multiplication. Call the results $R_k = M(x_k), R_{k+1} = M(x_{k+1}) M(x_k), \ldots, R_{k+N-1} = M(x_{k+N-1}) \cdots M(x_k)$.

3) For $i = k, \ldots, k + N - 1$, let output $y_i$ be the first element of $R_i$.

4) Update $s$ to be $R_{k+N-1}$.

Note that although the matrices $M$ are sparse, in general the matrices $R$ will have full upper triangles. Vector extensions are generally designed so that the dot products in the inner loops of matrix multiplications will vectorize, so this method should vectorize on such processors.

IV. Example 2: Finite State Machines

Let $M = (\Sigma, S, T, s_0, A)$ be an FSM with alphabet $\Sigma$, state set $S$, transitions $T : \Sigma \times S \rightarrow S$, initial state $s_0 \in S$, and accepting states $A \subseteq S$. Let’s assume again that an actor implementing this FSM is scheduled to run $N$ times consecutively with inputs $x_i \in \Sigma$, for $i \in k, \ldots, k + N - 1$ yielding corresponding boolean outputs $y_i$ which are true if and only if $M$ is in an accepting state after reading $x_i$. I will show two parallel scan-based algorithms for computing the outputs of the actor. The first one will be given by exhibiting a state vector $s$ and matrices $M(x_i)$ so that the algorithm presented in the previous section can be re-used. The second one does some pre-computation given $M$ so that table look-ups substitute for the matrix multiplications.

Without loss of generality, identify the $|S|$ states with the integers $1, \ldots, |S|$, with 1 being the initial state and the accepting states the highest-numbered $|A|$ states. In the vector-matrix algorithm, $s$ is a column vector of length $|S|$ containing a one at the index corresponding to the current state of $M$ and all other values zero. The initial value of $s$ is $s_0 = (1, 0, \ldots, 0)^T$. Let $M(x_i)$ contain a 1 at index $(r, c)$ if $T(x_i, c) = r$ and zero otherwise. Then as required for a representation of the transition function, if the machine is in state $c$ before reading symbol $x_i$, then $s_i = M(x_i) s_{i-1}$ is in state $r$, i.e., has its 1 at index $r$. Matrix multiplication is again the representation of composition of the transition functions, and the algorithm of the previous section can be applied, changing only the step where the outputs are computed:

1) Compute $M(x_k), \ldots, M(x_{k+N-1})$ in parallel.

2) Apply the scan operation to the matrices, where the binary operator is matrix multiplication. Call the results $R_k = M(x_k), R_{k+1} = M(x_{k+1}) M(x_k), \ldots, R_{k+N-1} = M(x_{k+N-1}) \cdots M(x_k)$.

3) For $i = k, \ldots, k + N - 1$, let output $y_i$ be true if and only if $R_i$ has its 1 at one of the $|A|$ highest indices.

4) Update $s$ to be $R_{k+N-1}$.

In contrast to the case of IIR filters, with FSMs the matrices $R$ are sparse: like the ‘$s$’s, they will have only one nonzero per row, and that element’s value must be a 1. All vectors $s$ likewise contain all zeros except for one element with value 1. So, the matrices $M$ and $R$ could be stored as a vectors of length $|S|$ containing the index of the 1 in each row and the vectors $s$ could be stored as the index of the element with value one. Matrix-matrix multiplication then would take $O(|S|^2)$ time and matrix-vector multiplication $O(1)$ time.

Further speed improvement is possible by doing some computation once the FSM is known but before the first invocation of the actor. As the matrices $M$ and $R$ contain only 1’s and 0’s, there can be only a finite number of them. So all the possible products $M(x_i) \cdots M(x_{j})$ of matrices
M(x) for x ∈ Σ can be carried out in advance and a table built of the results. Each of these matrices can be identified with an integer, so functional composition is no longer matrix multiplication but a look up of an entry in a 2D integer array. (Those knowledgeable in abstract algebra will recognize that this lookup table is isomorphic to the transition semigroup of the FSM. [17]) Functional composition now takes O(1) time. For details of this representation, of the resulting greedy algorithm for constructing the look up table, and a study of look up table size as a function of FSM size for one particular signal processing application see [18]. Previous table-driven methods yielding unbounded concurrency in FSM execution are described in [19].

V. EXPERIMENTAL VERIFICATION

References [18] and [20] present applications verifying the correctness and studies of the performance of the FSM method running on GPUs and on multicore CPUs. On GPUs, the look up tables of the previous section were stored in constant memory, which permits irregular, simultaneous access from multiple threads. Those studies did not consider the question of vectorization.

The vectorizability of the look up table method of the previous section was studied by coding the scan algorithm in C and compiling it with the Intel C++ Composer XE for Windows version 2013 SP1 vectorizing compiler targeting the AVX vector extensions. The inner loop of the scan could not be vectorized. Vectorizing this loop requires simultaneous reads of random addresses within the lookup table. Examination of the AVX vector extension instruction set [3] shows that it and earlier x86 vector extensions support only simultaneous reads of adjacent memory addresses. Thus, the table lookup is not vectorizable. The compiler identifies the entire loop as not being efficiently vectorizable. AVX-512 supports gather instructions that simultaneously read randomly indexed values into a vector register [3]. Hence this limitation is probably not permanent—but this assertion remains to be tested.

A software implementation of a simple SDF system was constructed in the Python 3 language. The IIR filter actor was implemented in OpenCL 1.2 using the parallel scan operator provided by PyOpenCL version 2013.2. Testing was performed on an Apple MacBook Pro with a dual core Intel i5-2435M CPU at 2.40 GHz and Apple’s OpenCL library. Figure 1 shows the output of the IIR filter y[n] = 0.99y[n − 1] + 0.01x[n] when excited with an impulse, yielding the expected exponential decay, RC filter response.

Actor execution times were measured using OpenCL profiling events inserted into the command queue before and after execution of the scan kernels. Figure 2 shows repeated measurements of the throughput of the IIR filter actor as a function of N. The peak throughput obtained is about 88 megasamples per second, on a relatively low power processor. The throughput increases with N up to approximately 100,000 samples. There is a classic trade-off between throughput and latency.

VI. CONCLUSIONS

Full use of the parallel computation capabilities of present and expected CPUs and GPUs require use of vector extensions. Yet many actors in data flow systems for digital signal processing have internal state that impose serial dependencies between actor invocations that make vectorizing across actor invocations impossible. We have presented a novel methodology that in some cases permits such vectorization through use of the parallel scan pattern. We presented examples of the use of the methodology on two types of actor useful in practice, along with experimental evaluation of the resulting actors.

REFERENCES

[1] W.-M. W. Hwu, GPU Computing Gems Jade Edition. Morgan Kauffman, 2012.
[2] J. Rupley, J. King, E. Quinell, F. Galloway, K. Patton, P. Seidel, J. Dinh, H. Bui, and A. Bhowmik, “The floating-point unit of the Jaguar x86 core,” in Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on, 2013, pp. 7 – 16.
[3] Intel architecture instruction set extensions programming reference. [Online]. Available: https://software.intel.com/en-us/intel-isa-extensions
[4] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, “Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms,” in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. IEEE, 2013, pp. 1107–1116.
[5] G. Zambusch, “Tuning a finite difference computation for parallel vector processors,” in Parallel and Distributed Computing (IPDPS), 2012 11th International Symposium on. IEEE, 2012, pp. 63–70.
[6] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, “An evaluation of vectorizing compilers,” in Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. IEEE, 2011, pp. 372–382.
[7] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[9] J. H. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim, “OpenCL performance evaluation on modern multi core CPUs,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1177–1185.

[10] S. Ritz, M. Pankert, and H. Meyr, “Optimum vectorization of scalable synchronous dataflow graphs,” in *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[11] R. E. Ladner and M. J. Fischer, “Parallel prefix computation,” *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[12] G. E. Blelloch, “Scans as primitive parallel operations,” *Computers, IEEE Transactions on*, vol. 38, no. 11, pp. 1526–1538, 1989.

[13] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, “Fast scan algorithms on graphics processors,” in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS ’08. New York, NY, USA: ACM, 2008, pp. 205–213. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375559

[14] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for GPU computing,” in *Graphics Hardware*, vol. 2007, 2007, pp. 97–106.

[15] N. Bell and J. Hoberock, “Thrust: A productivity-oriented library for CUDA,” in *GPU Computing Gems Jade Edition*, W. W. Hwu, Ed. Morgan Kaufmann, 2011, ch. 26, pp. 359–371.

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[17] S. Eilenberg, *Automata, Languages, and Machines*. Academic Press, 1976, vol. B, ch. 1, 6.

[18] L. Barford, “Parallelizing small finite state machines, with application to pulsed signal analysis,” in *Proc. IEEE Intl. Instrumentation and Measurement Technology Conf.*, May 2012, pp. 1957–1962.

[19] H. Lin and D. G. Messerschmitt, “Finite state machine has unlimited concurrency,” *IEEE Transactions on Circuits and Systems*, vol. 38, no. 5, pp. 465–475, 1991.

[20] L. Barford and K. Keenan, “Segmenting a signal based on a local property using multicore processors,” in *Proc. IEEE Intl. Instrumentation and Measurement Technology Conf.*, May 2014, pp. 397–401.