J Sign Process Syst (2017) 87:21–31
DOI 10.1007/s11265-015-1045-x

Data Flow Algorithms for Processors with Vector Extensions

Handling Actors With Internal State

Lee Barford[1] · **Shuvra S. Bhattacharyya**[2,3] · **Yanzhou Liu**[2]

Received: 26 February 2015 / Revised: 1 September 2015 / Accepted: 14 September 2015 / Published online: 4 November 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract Full use of the parallel computation capabilities of present and expected CPUs and GPUs requires 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 and one class of FSMs are studied.

✉ Lee Barford
lee.barford@keysight.com

1 Keysight Laboratories, Keysight Technologies, Inc., 561 Keystone Ave. Unit 434, Reno, NV 89503, USA

2 University of Maryland, College Park, MD, USA

3 Tampere University of Technology, Tampere, Finland

Keywords Digital signal processing · Data flow computing · Vector processors · Parallel algorithms · Graphics processing units

1 Introduction

Nearly a decade ago, microprocessor clock rates stopped increasing and the number of cores started increasing. Thought leaders in computer architecture projected that due to power scaling issues, the number of cores per microprocessor would increase exponentially. Microprocessors would consist of a large number of homogeneous processors, synchronizing and communicating through shared memory. The number of processors would double approximately every two years. Between 16 and 32 cores per microprocessor should have been common in 2015. A key question was how all of this parallelism would be used and effectively and efficiently programmed [2]. In the field of signal processing, many authors developed highly parallelizable methods, e.g. [3, 12, 31].

That projected future would have been an ideal one for application of data flow. Early workers in data flow wrote that a reason for using data flow was to expose and efficiently use parallelism in a platform that was relatively easy to program for general numeric computing [8] and also for signal processing [27]. Synchronous data flow (SDF) was in part conceived to address the problem of correctly and efficiently scheduling large-grain signal processing computations on homogeneous multiprocessors. Finer-grained data flows could be used within actors if it were desired to reveal parallelism at lower levels of data abstraction [18].

A few microprocessors intended for use in supercomputing and the highest performance applications of signal

processing are available with over 50 cores. They cost thousands of dollars and consume hundreds of watts. On the other hand, nearly all currently available processors with performance, cost, and power specifications suitable for most signal processing and embedded uses have at most four cores. At that low amount of parallelism, the problem of identifying sufficient parallelism failed to appear in most applications. In particular, SDF and its derivatives are often sufficient to identify and effectively use that level of coarse-grained task parallelism.

However, few 16+ core processors are available and they are not suitable for embedded applications. What happened? Moore's Law continued to double the number of transistors available to microprocessor designers. Each computation with that increased number of transistors continued to consume exponentially less power, so that total power consumption remained acceptable [15]. However, those transistors were not used solely, or even primarily, to increase the number of cores. Instead, from the programmer's point of view, many of them were spent on providing increased data rather than increased task parallelism.

This data parallelism has come in two forms.One form is the use of a graphics processor unit (GPU) for compute purposes [13]. 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) [1, 29] 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 [22]. 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 [35], 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 [21].

The third option is to write the most compute-intensive loops in a kernel language such as OpenCL [23] or CUDA [24]. 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 [19]. 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 items process the N data items, with no synchronization or passing of data required between them.

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 ith 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 [28].

2 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 $\mathbf{x}_1, \mathbf{x}_2, \ldots$ be the scalar- or vector-valued inputs to the actor at the first, second, and so on invocation of the actor. Let $\mathbf{s}_0$ be the initial values of the variables local to the actor that represent its internal state. Let $\mathbf{s}_i$ represent the values of those variables after the actor reads and processes inputs $\mathbf{x}_i$. Without 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 $\mathbf{s}_i = f(\mathbf{s}_{i-1}; \mathbf{x}_i)$ Or, once input $\mathbf{x}_i$ is available, we can write $\mathbf{s}_i = f_i(\mathbf{s}_{i-1})$ where $f_i(\mathbf{s}) = f(\mathbf{s}, \mathbf{x}_i)$. Now suppose that the actor is scheduled to run N times successively and the first invocation of this block of N is the kth one during the program execution. Then the successive values of the state variables are $f_k(\mathbf{s}_{k-1})$, then $(f_{k+1} \circ f_k)(\mathbf{s}_{k-1})$, then $(f_{k+2} \circ f_{k+1} \circ f_k)(\mathbf{s}_{k-1})$, and so on, where $\circ$ represents functional composition. The final value of the state variables is $(f_{k+N-1} \circ \ldots \circ f_k)(\mathbf{s}_{k-1})$. 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 [17], also called parallel scan [7], pattern can be applied to yield a parallel algorithm for computing all of the required functional compositions at once. Given objects $a_0, a_1, \ldots, a_{n-1}$ from a set C and an associative binary operation $\oplus$ on C, scan computes $\text{scan}(\oplus, a_0, \ldots, a_{n-1}) = (a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \ldots \oplus a_{n-1})$. (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 $a \in C$.)

A scan can be performed by a parallel recursion. In this method, the input is divided into two portions. The scan of the first portion is the corresponding portion of the whole output. The scan of the second portion is "almost" the second portion of the result: it is wrong in that the scan of the second portion starts adding from the first element of the second portion, but the correct scan of the second portion needs to have the sum of the elements in the first portion added in also. That is, to compute $\text{scan}(\oplus, x_0, \ldots, x_{n-1})$, perform the following (for simplicity of presentation, assume n is a power of 2):

1. (Base case) If $n = 1$ return the single-element array (x_0). If $n = 2$, return the two-element array $(x_0, x_0 \oplus x_1)$. Otherwise, continue with the following steps.
2. (Parallel recursion) Compute in parallel $T = \text{scan}(\oplus, x_0, \ldots, x_{n/2-1})$ and $U = \text{scan}(\oplus, x_{n/2}, \ldots, x_{n-1})$.
3. (Correct starting value of second segment) In parallel, add the final value of T to each element of U, that is, set U_i to $T_{n/2-1} \oplus U_i$ for $i \in \{n/2, n/2+1, \ldots, n-1\}$.
4. (Output) Concatenate the two arrays, T and the modified U, to produce the result S.

Implementation details of the scan algorithm are beyond the scope of this article: they may be found in [7, 17, 30]. The properties of scan that are important for the present purposes are:

– When there are sufficient processors or parallel lanes in vector mode instructions, the parallel execution of the scan requires $O(\lg n)$ time using $O(n)$ calls to the $\oplus$ function.
– There are a number of implementations of scan available in kernel languages [6, 9, 14, 30]. 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 the 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 $\mathbf{s}$.
2. Identify a representation for the state transformations f_i, including how to create f_i given actor input $\mathbf{x}_i$. 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 with net speedup will result.

It has been previously noted that parallel scan can be used automatically to find instruction-level parallelism in serial code. For example, Vishkin [32] proposed an instruction decoding scheme for serial processors where striding indexing operations would be replaced with instructions indicating pieces of single parallel scan to be performed. The resulting instruction stream would have dependencies between iterations of the same loop broken. This would increase the amount of available instruction level parallelism and thereby increase the usage of parallel functional units within a serial processor.

One way to view this methodology is as a software analogue of the look-ahead and relaxed look-ahead methods used to increase pipeline parallelism in some VLSI DSP implementations [26]. Another way to view this methodology is as engineering a semigroup (that is, a set and associated closed and associative binary operation) such that repeated use of the operation results in the desired computation.

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

3 Example 1: IIR Filters

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

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

The state required to compute the succeeding output $y[i]$ contains the previous Q outputs and $P-1$ previous inputs, or as a vector, $\mathbf{s}_{i-1} = (y[i-1], y[i-2], \ldots, y[i-Q], x[i-1], x[i-2], \ldots, x[i-P], 1)^T$. (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 $\mathbf{M}(x_i)$, a function of x_i,

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

The rth row of $M(x_i)$ tells how to compute the value of the rth state variable after the ith firing of the IIR filter actor. 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., $\mathbf{s}_i = \mathbf{M}(x_i)\mathbf{s}_{i-1}$. This matrix representation of an IIR filter is closely related to those used in CAD software that automatically produces an array of second order sections for VLSI implementation of an IIR filter from the filter coefficients and other circuit parameters such as the numeric word size [33]. 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 $\mathbf{M}$. That is, for any $m > 0$, $f_{k+m} \circ \ldots \circ f_k = \mathbf{M}(x_{k+m})\mathbf{M}(x_{k+m-1}) \cdots \mathbf{M}(x_k)$.

This yields the following algorithm for an IIR filter actor. At application start time, the state vector $\mathbf{s}$ 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 $\vec{s}$ are computed thus:

1. Compute $\mathbf{M}(x_k), \ldots, \mathbf{M}(k_{k+N-1})$ 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 $\mathbf{R}_k = M(x_k)$, $\mathbf{R}_{k+1} = \mathbf{M}(x_{k+1})\mathbf{M}(x_k), \ldots, \mathbf{R}_{k+N-1} = \mathbf{M}(x_{k+N-1}) \cdots \mathbf{M}(x_k)$.
3. For $i = k, \ldots, k+N-1$, let output y_i be the first element of $\mathbf{R}_i\mathbf{s}$.
4. Update $\mathbf{s}$ to be $\mathbf{R}_{k+N-1}\mathbf{s}$.

Note that although the matrices $\mathbf{M}$ are sparse, in general the matrices $\mathbf{R}$ 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.

4 Example 2: Finite State Machines

Let $M = (\Sigma, S, T, s_0, A)$ be an FSM with alphabet Σ, state set S, transitions $T : \Sigma \times S \to S$, 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. We 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 $\mathbf{s}$ and matrices $\mathbf{M}(x_i)$ 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 being the highest-numbered $|A|$ states. In the vector-matrix algorithm, $\mathbf{s}$ 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 $\mathbf{s}$ is $s_0 = (1, 0, \ldots, 0)^T$. Let $\mathbf{M}(x_i)$ 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, that is, $\mathbf{s}_{i-1}$ has its one at index c, then $\mathbf{s}_i = \mathbf{M}(x_i)\mathbf{s}_{i-1}$ 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 $\mathbf{M}(x_k), \ldots, \mathbf{M}(k_{k+N-1})$ in parallel.
2. Apply the scan operation to the matrices, where the binary operator is matrix multiplication. Call the results $\mathbf{R}_k = \mathbf{M}(x_k), \mathbf{R}_{k+1} = \mathbf{M}(x_{k+1}))\mathbf{M}(x_k), \ldots, \mathbf{R}_{k+N-1} = \mathbf{M}(x_{k+N-1}) \cdots \mathbf{M}(x_k)$.
3. For $i = k, \ldots, k+N-1$, let output y_i be true if and only if $\mathbf{R}_i\mathbf{s}$ has its 1 at one of the $|A|$ highest indices.
4. Update $\mathbf{s}$ to be $\mathbf{R}_{k+N-1}\mathbf{s}$.

In contrast to the case of IIR filters, with FSMs the matrices $\mathbf{R}$ are sparse: like the $\mathbf{M}$'s, they will have only one nonzero per row, and that element's value must be a 1. All vectors $\mathbf{s}$ likewise contain all zeros except for one element with value 1. So, the matrices $\mathbf{M}$ and $\mathbf{R}$ could be stored as a vectors of length $|S|$ containing the index of the 1 in each row and the vectors $\mathbf{s}$ could be stored as the index of the element with value one. Matrix-matrix multiplication then would take $O(|S|)$ 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 $\mathbf{M}$ and $\mathbf{R}$ 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 $\mathbf{M}(x)$ for $x \in \Sigma$ 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. 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 [4]. Previous table-driven methods yielding unbounded concurrency in FSM execution are described in [20].

4.1 Previous Work Relating Finite Semigroups and FSMs

Those knowledgeable in abstract algebra will recognize that the lookup table described above is isomorphic to the transition semigroup of the FSM [11]. In other words, what we have shown is that the operation of an FSM can be computed with a cumulative sum with its operation provided by a finite semigroup that can be found constructively from the FSM.

The inverse is also true. An FSM M' that computes the cumulative sum of the finite semigroup with elements S and operation $\oplus$ is constructed as follows. The states and input alphabet of M' are both S. When symbol $s \in S$ is input, a transition is made from state t to state u where $u = t \oplus s$. At any stage of reading its input, the state of M' is the cumulative sum under the semigroup operation of the prior inputs.

Knowledge of this correspondence between FSMs and finite semigroups is not new. In the early 1960's, this correspondence was used to develop Krohn-Rhodes theory, including the proof that all finite state machines can be implemented (under a suitable formal notion of what "implemented" means) as a combination of circuits comprising counters and combinatorial circuits. The key theorem shows that the semigroup corresponding to the FSM can be decomposed into a suitably defined product of groups (the counters) operated on by functions corresponding to the combinatorial circuits. [16] This decomposition is not practical, both in the sense that the decomposition process itself is not computationally tractable and also in the sense that the decomposed structures might require many times more logical and mathematical operations to execute than would be orginal FSM.

Work in the 1960's and 70's characterized which FSMs could be practically decomposed algebraically [25].

Work continues in defining new notions of a “product” of algberaic structures, seeking practical algebraic decompositions of the semigroups corresponding to FSMs, e.g. [10, 34].

5 Experimental Verification

5.1 Finite State Machines

References [4] and [5] 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 [1] 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 [1]. Hence this limitation is probably not permanent—but this assertion remains to be tested.

On GPUs, code without if-statements and with few automatic variables necessarily vectorizes, with at least a warp of threads executing in a single-instruction, multi-thread fashion. The inner loop of the parallel prefix computation of the FSM method presented above consists of a single read from a table, so it will in this sense vectorize. GPUs have a complex memory model, with different sorts of memory offering various tradeoffs between generality of use and performance. For example, many recent NVIDIA GPUs provide a cache for read-only data with each execution unit (streaming multiprocessor, or SM) that are not kept coherent with global memory writes or with one another. The working set size of these constant memory caches is small, under 50 kilobytes. Since the semigroup table is fixed for a particular FSM, it should be cached in the read-only caches. Hence, a key question for the performance of the proposed FSM method on GPUs is: How will the performance vary with the size of the semigroup table?

One experimental approach to this question would be to select a set of FSMs of increasing size and complexity and measure their performance on the same GPU(s). However, as there is no straightforward relationship between FSM size and semigroup table size, the semigroup table sizes tested would be uneven.

Instead, we will measure the performance of the proposed method on a set of FSMs that have semigroup tables of predictable size that are easily constructed. These are the counter-reset machines. These machines have two input symbols, here represented **a** and **b**. The machines count the number of **a**’s read until some maximum N is reached. The count is reset to 0 whenever a **b** is read. A counter-reset machine with $N = 5$ is illustrated in Fig. 1. As we will see below, the transition semigroup for the counter-reset machine for a given N has $2N$ elements. So it is possible to test the performance of the proposed method with any semigroup table size with even dimensions.

The transition semigroup of a counter-reset machine with maximum count N has $2N$ elements. We will write them $\mathbf{set}(0), \mathbf{set}(0), \ldots, \mathbf{set}(N)$ and $\mathbf{add}(1), \mathbf{add}(2), \ldots, \mathbf{add}(N-1)$. Recall that elements of the semigroup are tranformations over FSM states. So, to define each of the semigroup elements we need to say how it operates as a mapping from states to states. Then we can find the semigroup operation by computing the compositions of these mappings. Let

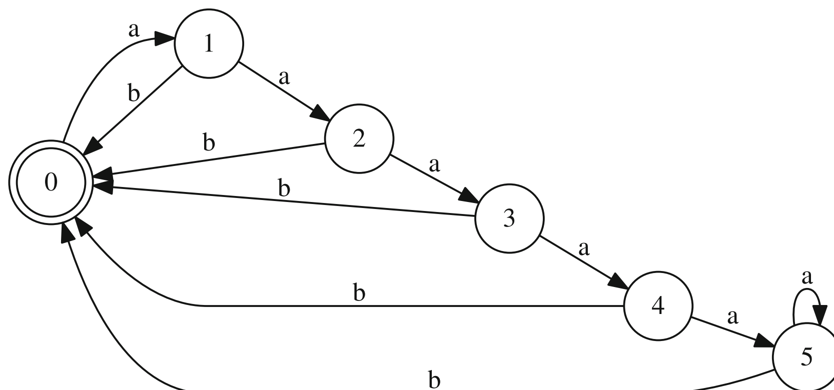

Figure 1 Counter-reset machine with $N = 5$.

Table 1 Semigroup table for a count-reset FSM that counts up to 5 shown in Fig. 1. Integers 0 through 5 represent the **set** transformations and 6 though 9 the **add** transformations.

$\oplus_{CR5}$	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	1	2	3	4
1	0	1	2	3	4	5	2	3	4	5
2	0	1	2	3	4	5	3	4	5	5
3	0	1	2	3	4	5	4	5	5	5
4	0	1	2	3	4	5	5	5	5	5
5	0	1	2	3	4	5	5	5	5	5
6	0	1	2	3	4	5	7	8	9	5
7	0	1	2	3	4	5	8	9	5	5
8	0	1	2	3	4	5	9	5	5	5
9	0	1	2	3	4	5	5	5	5	5

set(0) map all states to state 0. That is, **set**(0) is the reset operation that occurs when symbol **b** is read. Similarly, **set**(1) maps all states to state 1, **set**(2) maps all states to state 2, and so on. The semigroup elements **add**(i) map state j to state $\max(i + j, N)$, for $i = 1, \ldots, N - 1$. The semigroup element **add**(1) is the element that represents the reading of symbol **a**.

The composition of these transformation are as follows. For any semigroup element x, $i \in \{0, \ldots, N\}$, and $j, k \in \{1, \ldots, N - 1\}$,

$$x \circ \mathbf{set}(i) = \mathbf{set}(i), \tag{3}$$

$$\mathbf{set}(i) \circ \mathbf{add}(j) = \mathbf{set}(\min(i + j, N)), \tag{4}$$

and

$$\mathbf{add}(j) \circ \mathbf{add}(k) = \begin{cases} \mathbf{add}(j+k) & \text{if } j+k < N \\ \mathbf{set}(N) & \text{otherwise.} \end{cases} \tag{5}$$

It is natural to store the semigroup elements as integers. In what follows we represent **set**(i) with i and **add**(i) with $i + N$.

Table 1 gives the semigroup table for the FSM of Fig. 1. The integers 0 through 5 represent the transformations **set**(0) through **set**(5). The integers 6 though 9 represent the transformations **add**(1) through **add**(5). The semigroup element input to the parallel scan to indicate reading of symbol **a** is **add**(1), represented here by the integer 0.

Again, the purpose of generating a table for a counter-reset FSM is to test the speed of the proposed method with abitrarily sized semigroup tables. It would probably be faster and certainly use less memory to implement a counter-reset FSM using Eqs. 3–5 directly.

The performance of the proposed method on counter-reset FSMs with various values of N was measured using an NVIDIA Geforce GTX 780 GPU programmed using CUDA 6.0. The parallel scan used was from the CUDA

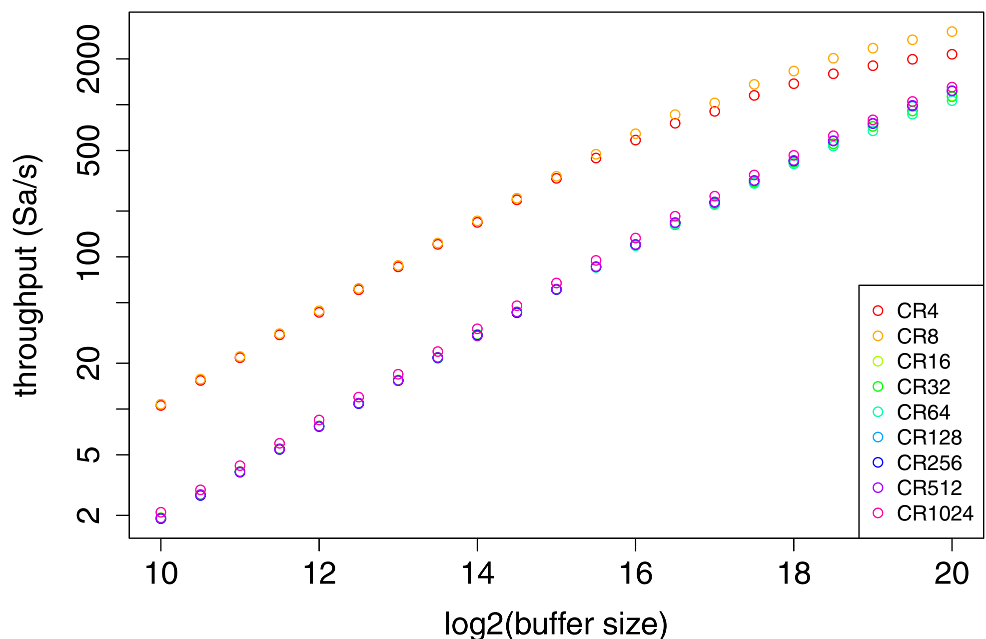

Figure 2 Throughput of counter-reset FSMs on a GPU as a function of buffer length (log-log plot). Color indicates the value of N for the counter-reset machine. "CRn" indicates a counter-reset machine with $N = n$. Each plotted datum is the median of 20 repeated runs.

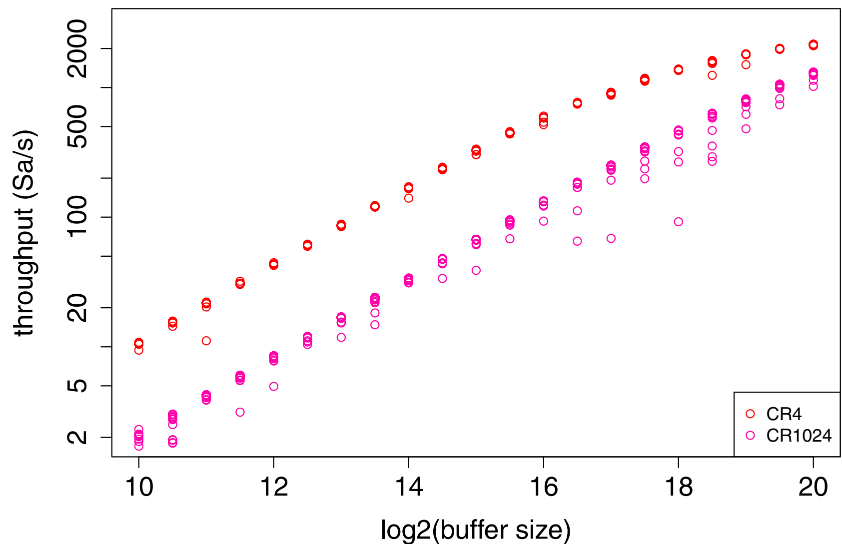

Figure 3 Throughput of counter-reset FSMs on a GPU as a function of buffer length (log-log plot). Color indicates the value of N for the counter-reset machine. All repeated runs are plotted for the count-reset machines with $N = 4$ and $N = 1024$, labelled CR4 and CR1024, respectively.

Thrust [6] C++ template library version 1.7. In particular, the scan algorithm `transform_exclusive_scan` was used because this algorithm combines a transformation of the input vector into the first pass of the parallel scan. Here, that transformation is the mapping of the input symbols to their corresponding semigroup elements, eliminating one pass through the inputs. This increases the computational intensity, that is, the number of computational operations per memory access. Sixteen bit integers were used to represent the semigroup elements.

Data transfers between the CPU and GPU were not included in execution timings. There are two reasons for this. First, in practice the FSM is unlikely to be the only actor implemented on the GPU. Other operations are likely to be performed on the GPU before the FSM, such as calibration correction, clock recovery or demodulation, and decoding from real or complex voltages to symbols. The second reason is that in CPU-GPU hybrid chips, the CPUs and GPU share a global memory, eliminating the data copying that is necessary when the GPU is on a daughter card residing on an input/output bus.

The throughput of the FSMs on this GPU are shown in Fig. 2. In this log-log plot, each point plotted is the median of 20 repeated measurements. Median rather than mean was chosen because of its relative insensitivity to outliers. There are three notable features of this plot. First, increasing buffer size results in significant increase in throughput. Since increasing buffer size induces a corresponding increase in latency, this means that there is a significant throughput versus latency design tradeoff to be made when selecting the buffer size. Another feature of interest is that the FSMs with small semigroup table sizes ($N = 4$ and $N = 8$, with 8x8 and 16x16 semigroup tables, respectively) had throughput approximately 2-5 times that of the best performing of the FSMs with larger semigroup tables. Another notable feature of the plot is that (except for $N = 4$ and $N = 8$) FSMs with larger sermigroup tables have higher throughput than those with smaller tables.

In order to investigate the significance of these these last two phenomena, in Fig. 3 are plotted all of the measured data for the FSMs with $N = 4$ and $N = 1024$. These two machines are those on either side of the "gap" in throughput apparent in Fig. 2. Although most runs had throughput near the median, there are a number of lower throughput outliers. The spread of these outliers is small compared to the gap between the throughput for machines with small and larger semigroup tables. The spread is noticeably larger, however, than the differences among median throughputs in Fig. 2.

The reason for this complex behavior as a function of semigroup table size is unclear. However, as the code is in all cases identical and only the size of this table changes, cache effects are implicated.

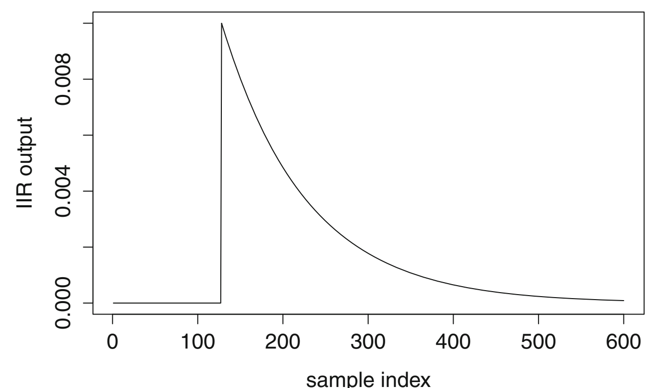

Figure 4 Output of IIR filter. Input is a unit impulse at sample index 128.

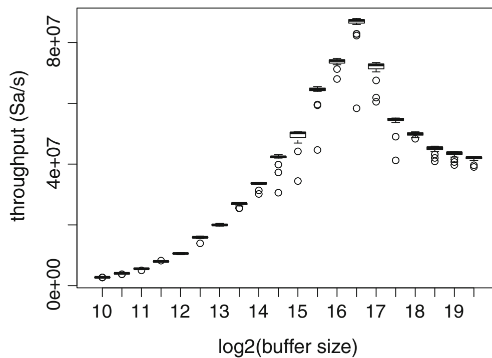

Figure 5 Boxplot of 20 repeated measurements of the base 2 logarithm of the number of consecutive actor invocations N (in this implementation the same as the size of sample buffers passed to the actor) vs throughput in samples per second of the IIR filter.

5.2 IIR Filters

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 4 shows the output of the IIR filter $y[n] = 0.99\, y[n-1] + 0.01\, x[n]$ 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 5 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.

6 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. For both actors, there were substantial tradeoffs to be made between throughput and latency by selecting buffer sizes.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Intel architecture instruction set extensions programming reference. https://software.intel.com/en-us/intel-isa-extensions.
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., & et al. (2006). *The landscape of parallel computing research: A view from Berkeley. Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department*. Berkeley: University of California.
3. Barford, L. (2011). Speeding localization of pulsed signal transitions using multicore processors. *IEEE Transactions Instrumentation and Measurement*, *60*(5), 1588–1593.
4. Barford, L. (2012). Parallelizing small finite state machines, with application to pulsed signal analysis. In *Proc. IEEE Intl. Instrumentation and Measurement Technology Conf.* (pp. 1957–1962).
5. Barford, L., & Keenan, K. (2014). Segmenting a signal based on a local property using multicore processors. In *Proc. IEEE Intl. Instrumentation and Measurement Technology Conf.* (pp. 397–401).
6. Bell, N., & Hoberock, J. (2011). Thrust: A productivity-oriented library for CUDA. In Hwu, W.W. (Ed.) *GPU Computing Gems Jade Edition, chap. 26* (pp. 359–371). Morgan Kauffman.
7. Blelloch, G.E. (1989). Scans as primitive parallel operations. *Computers, IEEE Transactions on*, *38*(11), 1526–1538.
8. Dennis, J.B. (1980). Data flow supercomputers. *IEEE Computer*, *13*(11), 48–56.
9. Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., & Manferdelli, J. (2008). Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS '08* (pp. 205–213). New York: ACM. doi:10.1145/1375527.1375559.
10. Egri-Nagi, A., Mitchell, J.D., & Nehaniv, C.L. (2014). SgpDec: Cascade (de)composition of finite transformation semigroups and pertmutation groups. In Hong, H., & Yap, C. (Eds.) *Proc. of the 4th Internationcal Conference on Mathematical Software (ICMS 2014), Lecture Notes in Computer Science*, (Vol. 8592 pp. 75–82). Springer.
11. Eilenberg, S. (1976). *Automata, Languages, and Machines, vol. B, chap. 1, 6.*: Academic Press.
12. Franchetti, F., Voronenko, Y., & Puschel, M. (2006). FFT program generation for shared memory: SMP and multicore. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (pp. 51–51). IEEE.
13. Hwu, W.M.W. (2012). *GPU Computing Gems Jade Edition*. Morgan Kauffman.
14. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, *38*(3), 157–174.
15. Koomey, J.G., Berard, S., Sanchez, M., & Wong, H. (2011). Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, *33*(3), 46–54.

16. Krohn, K., & Rhodes, J. (1965). Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, *116*, 450–464.
17. Ladner, R.E., & Fischer, M.J. (1980). Parallel prefix computation. *Journal of the ACM*, *27*(4), 831–838.
18. Lee, E.A., & Messerschmitt, D.G. (1987). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions Computers*, *36*(1), 24–35.
19. Lee, J.H., Patel, K., Nigania, N., Kim, H., & Kim, H. (2013). OpenCL performance evaluation on modern multi core CPUs. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)* (pp. 1177–1185). IEEE.
20. Lin, H., & Messerschmitt, D.G. (1991). Finite state machine has unlimited concurrency. *IEEE Transactions on Circuits and Systems*, *38*(5), 465–475.
21. Maleki, S., Gao, Y., Garzaran, M.J., Wong, T., & Padua, D.A. (2011). An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011* (pp. 372–382). IEEE.
22. Mitra, G., Johnston, B., Rendell, A.P., McCreath, E., & Zhou, J. (2013). Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)* (pp. 1107–1116). IEEE.
23. Munshi, A., Gaster, B., Mattson, T.G., & Ginsburg, D. (2011). *OpenCL programming guide*: Pearson Education.
24. Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, *6*(2), 40–53.
25. Nozaki, A. (1978). Practical decomposition of automata. *Information and Control*, *36*, 275–291.
26. Parhi, K.K. (1995). High-Level algorithm and architecture transformations for DSP synthesis. *Journal of VLSI Signal Processing*, *9*, 1–2.
27. Radivojevic, I.P., & Herath, H. (1991). Executing DSP applications in a fine-grained dataflow environment. *IEEE Transactions on Software Engineering*, *17*(10), 1028–1041.
28. Ritz, S., Pankert, M., & Meyr, H. (1993). Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*.
29. Rupley, J., King, J., Quinnell, E., Galloway, F., Patton, K., Seidel, P., Dinh, J., Bui, H., & Bhowmik, A. (2013). The floating-point unit of the Jaguar x86 core. In *2013 21st IEEE Symposium on Computer Arithmetic (ARITH)* (pp. 7–16).
30. Sengupta, S., Harris, M., Zhang, Y., & Owens, J.D. (2007). Scan primitives for GPU computing. In *Graphics Hardware, vol. 2007* (pp. 97–106).
31. Tan, K., Liu, H., Zhang, J., Zhang, Y., Fang, J., & Voelker, G.M. (2011). Sora: High-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, *54*(1), 99–107.
32. Vishkin, U. (1997). From algorithm parallelism to instruction-level parallelism: An encode-decode chain using prefix-sum. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures* (pp. 260–271).
33. Wu, C.W., & Cappello, P.R. (1998). Application-specific CAD of VLSI second-order sections. IEEE Transactions on Acoustics. *Speech and Signal Processing*, *36*(5), 813–825.
34. Zeiger, H.P. (1968). *Algebraic Theory of Machines, Languages, and Semigroups, chap. Cascade decomposition of automata using covers*, (pp. 55–80). Academic Press.
35. Zumbusch, G. (2012). Tuning a finite difference computation for parallel vector processors. In *11th International Symposium on Parallel and Distributed Computing (ISPDC), 2012* (pp. 63–70). IEEE.

Lee Barford is a Fellow at Keysight Laboratories and an adjunct Professor of Computer Science and Engineering at the University of Nevada, Reno, Nevada. He received the Ph.D. degree in Computer Science from Cornell University in 1987. He leads Keysight's efforts in applying parallel computing to speed electronic measurement and test. He also leads research to identify and apply emerging technologies in software, applied mathematics, and statistics to enable new kinds of measurements and increase measurement accuracy and speed. Previously, he managed a number of research projects at Agilent Laboratories (Santa Clara, California) and Hewlett- Packard Laboratories (Palo Alto, California), for example in visible light and X-ray imaging systems, calibration methods for non-linear and dynamical disturbances, and fault isolation from automatic test equipment results. He is an author of over fifty papers and an inventor on over forty patents.

Shuvra S. Bhattacharyya is a Professor in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. He holds a joint appointment in the University of Maryland Institute for Advanced Computer Studies (UMIACS). He is also a part time visiting professor in the Department of Pervasive Computing at the Tampere University of Technology, Finland, as part of the Finland Distinguished Professor Programme (FiDiPro). He is an author of six books, and over 250 papers in the areas of signal processing, embedded systems, electronic design automation, wireless communication, and wireless sensor networks. He received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and Compiler Developer at Kuck & Associates (Champaign, Illinois). He has held a visiting research position at the US Air Force Research Laboratory (Rome, New York). He has been a Nokia Distinguished Lecturer (Finland) and Fulbright Specialist (Austria and Germany). He has received the NSF Career Award (USA). He is a Fellow of the IEEE.

Yanzhou Liu is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Maryland, College Park (UMCP). She received the bachelor's degree in Electronic and Information Science and Technology from Fudan University, Shanghai, China. She joined the Maryland DSPCAD Research Group as a Ph.D. Student in 2013 and is currently working toward Ph.D. degree in UMCP. She studied feature extraction and dimensionality reduction on hyperspectral images in Fudan University. Her research interests include model-based design in real-time instrumentation system, dataflow graph implementation in multicore system including GPU, hardware-software co-design and digital signal processing system. She is an author of two papers.