ADAPTIVE FILTERS USING MODIFIED SLIDING-BLOCK DISTRIBUTED ARITHMETIC WITH OFFSET BINARY CODING

Walter Huang, David V. Anderson

Georgia Institute of Technology
School of Electrical and Computer Engineering
Atlanta, GA 30332-0250

ABSTRACT

An efficient way for computing the response of an adaptive digital filter is to use sliding-block distributed arithmetic (SBDA). One disadvantage of distributed arithmetic is the amount of memory utilized. By encoding the memory tables in offset binary code (OBC), the size of the memory tables is reduced in half. However, the computational workload remains unchanged. By modifying the computational flow, the computational workload can be reduced by almost half at the expense of slightly more memory. This modified SBDA structure is called SBDA-OBC. It has memory requirements 25%-50% lower than SBDA depending on the size of the sub-filter. In terms of the computational workload, SBDA-OBC is most advantageous for large sub-filters and when the filter is split into few sub-filters. In this case, the computational workload is reduced almost in half.

Index Terms— adaptive filtering, distributed arithmetic

1. INTRODUCTION

A fundamental digital signal processing operation is filtering, and it is commonly computed using multipliers and adders. When computed sequentially, the multiplication of two B-bit numbers requires from $B/2$ to B additions, and is time intensive. Alternatively, the multiplication can be computed in parallel using $B/2$ to B adders, but is area intensive [1, 2]. Whether a K-tap filter is computed serially or in parallel, it requires at least $B/2$ additions per multiplication plus the $K-1$ additions for summing the products together.

A competitive alternative to using a multiplier is distributed arithmetic (DA). It compresses the computation of the filter into a memory table and generates a result in B-bit time using $B-1$ additions. DA significantly reduces the number of additions needed for filtering [3, 4]. Using a DA filter structure reduces the number of additions by a factor K at the expense of a 2^K word memory table. This reduction in the computational workload is a result of storing the pre-computed partial sums of the filter coefficients in the memory table [1].

However, filter structures that use DA are not well suited for adaptive applications. The primary advantage of DA is that pre-computed tables can be used to eliminate multipliers in the filtering operation; however, in an adaptive system, the contents of the tables must be continually recomputed. Regenerating the contents of the DA tables requires a significant number of computations, reducing or eliminating the computational advantage of DA. A couple of adaptive DA filters have been published to address this issue [5, 6, 7]. In these filter structures, the computational workload of the update was reduced by only recomputing a few memory table entries per sample period. However, this design choice results in a reduction in the convergence rate of an adaptive algorithm.

To avoid such a reduction, a filter structure that updates the contents of the entire memory table in a manner noticeably more efficient than brute force is required. A couple of such structures exist [1, 8]. One such structure is called sliding-block distributed arithmetic (SBDA) [1]. Although it uses one of the most efficient methods for updating its entire memory table, its performance can still be further improved.

In this paper, the focus is on modifying SBDA to significantly reduce the computational workload and to encode the memory tables in offset binary code (OBC). The background material on SBDA and the use of OBC for DA are contained in Sections 2 and 3, respectively. This is followed by a description of how SBDA is modified to use OBC while significantly reducing the computational workload over SBDA. Finally, the results are presented, and the advantages of this modified SBDA structure over the original one are highlighted.

2. SLIDING-BLOCK DISTRIBUTED ARITHMETIC

An efficient way for computing the response of an adaptive digital filter using DA is presented in [1]. This form of DA is called SBDA. It is a coefficient driven mechanization of DA. Since the contents of the memory tables are changing over time, the key issue is developing a method that minimizes the number of additions needed to build those tables. In SBDA, the input data is collected in blocks and then the appropriate samples are windowed and convolved with the FIR filter coefficients as shown in Fig. 1. This type of DA only changes the contents of one DA memory table every sample period. In SBDA, a K-tap filter is broken into $m+1$ k-tap sub-filters where $m = K/k$. Only the contents of the sub-filter with outdated samples need to be updated. If the update of that sub-filter were done using brute force, then it would take $(k/2-1)2^k+1$ additions. In SBDA, an observation is noted that the contents of the memory change slowly over time. In other words, only the oldest input sample needs to be removed while the newest input sample needs to be added. Thus, only 2^{k-1} additions and 2^{k-1} subtractions for a total of 2^k computations are required for a k-tap sub-filter. This results in a reduction in the number of operations necessary by about a factor of $k/2-1$ over brute force. However, in SBDA, the subtractions are eliminated by associating each DA memory table with a fixed set of inputs that further reduces the computational workload.

The computational workload for SBDA is split into performing two functions, updating and filtering. The normal order of operation is first to update and then to filter. The filtering function is done in a typical DA fashion [4]; hence, its operation is not described in this paper. The second function is for updating the memory table that contains the outdated input samples, and the modification of this

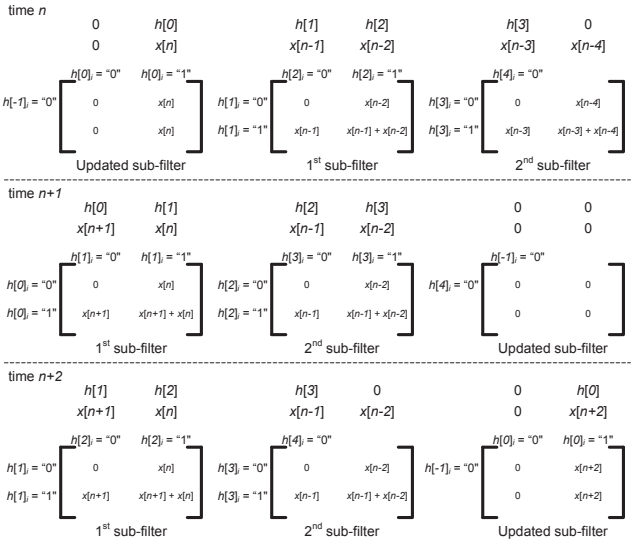

Fig. 1. An illustration of SBDA in action, where $K = 4$, $m = 2$, and $k = 2$.

update method is one of the primary focuses of this paper.

To reduce the computational workload for updating m memory tables, the objective of SBDA is to minimize the number of tables that need to be updated, in this case just one. This goal is accomplished by associating each sub-filter with a fixed set of k sequential input samples ordered in a sequential manner. Initially, it is assumed that all the input data is stored in the m sub-filters and that all the input samples are used. When a new sample arrives, it is not stored in any sub-filter. Instead of replacing the oldest sample with this sample, which would take 2^{k-1} additions to insert the new sample and 2^{k-1} subtractions to remove the old sample, a new sub-filter whose memory table has been initialized to zero can be created to store this new sample. With no outdated samples in the table, the update is simply to just add the new sample into the appropriate 2^{k-1} memory table locations. This process continues until the memory table is full. At this point in time, one of the sub-filters is completely outdated, and can be reused to create the new one.

This update process uses only 2^{k-1} additions and no subtractions per sample period, and further reduces the computational workload in half. However, this reduction is at the expense of an extra sub-filter, hence an extra memory table. Also, as a consequence, the filter coefficients may span more than m memory tables, and some input samples in the first and last sub-filters may not be used. Any unused sample is filtered with zero. An illustration of this update process in action is shown in Fig. 1.

3. DISTRIBUTED ARITHMETIC USING OFFSET BINARY CODING

OBC can be used to halve the size of the memory tables in DA [4]. The derivation for OBC begins with writing the input, $x[n-i]$, as follows.

$$x[n-i] = \frac{1}{2}\{x[n-i] - (-x[n-i])\}, \quad i = 0, \ldots, K-1 \quad (1)$$

Next, x_i and $-x_i$ are written in two's complement notation as shown in 2 and 3 respectively and substituted back into 1 to yield 4.

$$x[n-i] = -b_{i0} + \sum_{l=1}^{B-1} b_{il} 2^{-l} \quad (2)$$

$$x[n-i] = -\bar{b}_{i0} + \sum_{l=1}^{B-1} \bar{b}_{il} 2^{-l} + 2^{-(B-1)} \quad (3)$$

$$x[n-i] = \frac{1}{2}\left[-(b_{i0} - \bar{b}_{i0}) + \sum_{l=1}^{B-l} (b_{il} - \bar{b}_{il}) 2^{-l} - 2^{-(B-1)}\right] \quad (4)$$

Eq. 4 can be rewritten as Eq. 5 where the variable, c_{il}, is defined as $b_{il} - \bar{b}_{il}$.

$$x[n-i] = \frac{1}{2}\left[-c_{i0} + \sum_{l=1}^{B-1} c_{il} 2^{-l} - 2^{-(B-1)}\right] \quad (5)$$

Now, with $x[n-i]$ written in OBC notation, its incorporation into the FIR filtering equation is detailed below in Eq. 8 where $Q(b_l) = \sum_{i=0}^{K-1} \frac{h[i]}{2} c_{il}$ and $Q(0) = \sum_{i=0}^{K-1} \frac{-h[i]}{2}$.

$$y[n] = \sum_{i=0}^{K-1} h[i] x[n-i] \quad (6)$$

$$y[n] = \frac{1}{2} \sum_{i=0}^{K-1} h[i] \left[-c_{i0} + \sum_{l=1}^{B-l} c_{il} 2^{-l} - 2^{-(B-1)}\right] \quad (7)$$

$$y[n] = -Q(b_0) + \sum_{l=1}^{B-1} Q(b_l) 2^{-l} + 2^{-(B-1)} Q(0) \quad (8)$$

The reduction in memory comes at the expense of slightly more hardware. This trade-off is often worthwhile [4].

4. COMBINING SLIDING-BLOCK DISTRIBUTED ARITHMETIC WITH OFFSET BINARY CODING

By applying OBC to SBDA, the size of the memory tables can be reduced in half. However, this straight-forward combination of SBDA with OBC only yields a memory benefit. Still, an addition or a subtraction is required for each entry in the updated memory table [1]. For a memory table not encoded using OBC, SBDA uses the same number of operations because only half of the memory table, which is twice the size of the one used when it is encoded in OBC, needs to be updated. Therefore, the number of computations needed is equal to the amount used in SBDA without OBC.

Recall that in SBDA when a DA memory table is about to be updated, it is initialized to zero. To generate the tables in OBC format, half of the most current input, $x[n]$, needs to be added or subtracted from every entry based on the bit stream for the current sample, b_{0l}. When $b_{0l} =$ "0", $0.5x[n]$ is subtracted, and when $b_{0l} =$ "1", $0.5x[n]$ is added . As an alternative, a block of the k most current samples can be collected and used to compute the initial condition, $Q(0)$, so that the DA memory tables are initialized to it. With the exception of the first step where no update is required, each subsequent update only needs to add the current sample to the entries where $b_{0l} =$ "1", which maps only to half of the table. This modification of SBDA is called SBDA-OBC.

A comparison of how the computational flow for updating a DA memory table encoded using OBC for a 3-tap filter differs between SBDA and SBDA-OBC is shown in Fig. 2. For both SBDA and SBDA-OBC, three steps plus the initialization of the memory contents are required to completely update a memory table. One step is taken every sample period, and does not affect the filtering computation. For SBDA, eight additions are needed, and for SBDA-OBC, six additions are required. In the first step for SBDA, no additions

are needed because the current sample plus zero is equal to the current sample. For each subsequent step, four additions are used. In the first step for SBDA-OBC, no additions are needed because no update is necessary since the memory table is already initialized to the correct values. For each subsequent step, it uses two additions at every step. Plus, two additions for computing $Q(0)$, which is necessary to update the memory table. In this case, SBDA-OBC uses 25% fewer additions than SBDA.

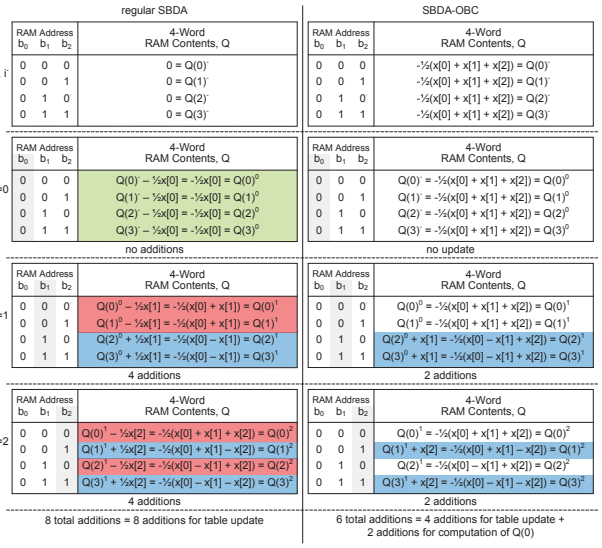

Fig. 2. A comparison of how SBDA and SBDA-OBC generates a DA memory table encoded in OBC.

5. COMPARISON OF SBDA AND SBDA-OBC

A logical starting point for making a comparison of SBDA and SBDA-OBC is to create formulas for certain critical measurement criteria. In this paper, the measurements of importance are the data memory usage and the computational workload required for the computation of one sample. Since in the case of SBDA and SBDA-OBC, the only type of computation of significance that is used is addition. The computational workload is measured as the number of additions needed for both updating the memory table and filtering the data. These four formulas are listed below.

$$SBDA_{additions} = B_h\left(\lfloor K/k \rfloor + 1\right) + \left(2^{k-1} - 1\right) \quad (9)$$

$$SBDA-OBC_{additions} = B_h\left(\lfloor K/k \rfloor + 1\right) + \left(2^{k-2} + 1\right) \quad (10)$$

$$SBDA_{memory} = 2^k\left(\lfloor K/k \rfloor + 1\right) \quad (11)$$

$$SBDA-OBC_{memory} = \left(2^{k-1} + 1\right) \cdot \left(\lfloor K/k \rfloor + 1\right) \quad (12)$$

where B_h is the bit precision of the filter coefficients, K is the filter length, and k is the size of the sub-filter. A table of the computational workload and of the data memory usage for different K when $B_h = 16$ and when k is selected such that the computational workload is minimized is given below in Table 1. If there are multiple configurations that minimize the computational workload, then the configuration with the lowest memory usage is reported.

To illustrate the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for the computation of one sample as B_h is varied and $k = 8$, a plot of this relationship is shown in Fig. 3. From this figure, it is observed that SBDA-OBC is most beneficial when the bit precision of the filter coefficients is low and the filter is grouped into few sub-filters. This advantage diminishes as the number of additions needed for updating becomes numerically insignificant to the number of additions needed for filtering when either B_h is large, the filter is split into many sub-filters (i.e. $\lfloor K/k \rfloor$ is large), or a combination of both.

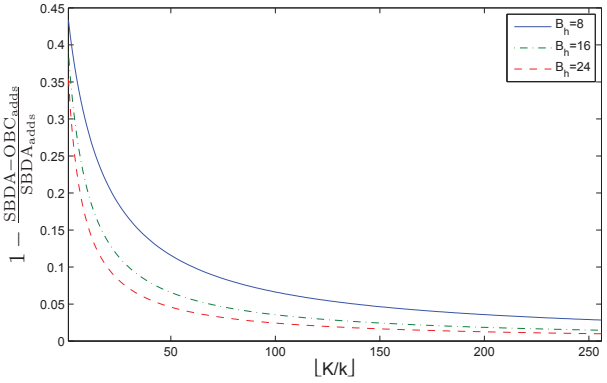

Fig. 3. A plot of the $1 - \frac{SBDA-OBC_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying B_h when $k = 8$.

Fig. 4 is a plot of the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for the computation of one sample when k is varied and $B_h = 16$. It is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks and the filter is grouped into few sub-filters. By increasing k, the ratio of computations used for updating over computations used for filtering increases; hence, the benefit of using SBDA-OBC over SBDA is significant.

An interesting observation is that SBDA-OBC is not beneficial when $k = 2$. In this case, the additional overhead associated with generating the initial condition, which is essential in the computational reduction of updating the memory table in other cases, in SBDA-OBC is significant. Specifically, in this case, SBDA requires one addition to update its memory table. However, SBDA-OBC requires two additions, one to update a memory table entry and another to compute the initial condition.

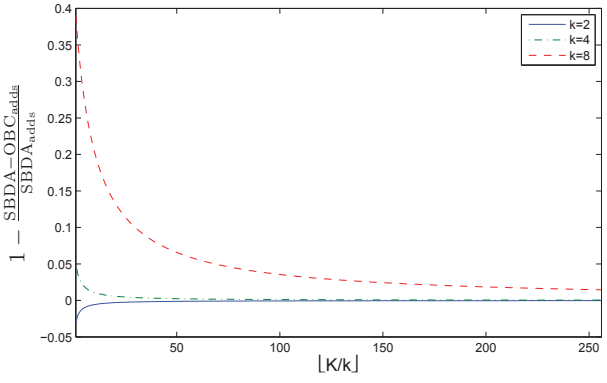

Fig. 4. A plot of the $1 - \frac{SBDA-OBC_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying k when $B_h = 16$.

A plot of the relative advantage of SBDA-OBC over SBDA in terms of the memory usage when k is varied is shown in Fig. 5. Note, this advantage is not dependent on K or on B_h. From the figure, it is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks. Although SBDA-OBC still has a 25% advantage over SBDA when k is small, this advantage is

Table 1. Computational Workload and Data Memory Usage for Various Filter Configurations when $B_h = 16$

	$k_{optimial}$*		# of Additions			# of Memory Words		
K	SBDA-OBC	SBDA	SBDA-OBC	SBDA-OBC**	SBDA	SBDA-OBC	SBDA-OBC**	SBDA
16	6	5	65	73	79	99	68	128
32	6	5	113	121	127	198	119	224
64	6	6	193	193	207	363	363	704
128	7	7	337	337	367	1235	1235	2432
256	8	7	593	625	655	4257	2405	4736

* $k_{optimal}$ is defined as the k that minimizes the computational workload.
** The value provided is for the SBDA-OBC filter configuration when the size of the sub-filter is set to the $k_{optimal}$ for SBDA.

diminished because of the slight memory overhead associated with SBDA-OBC. However, as the size of the memory tables increase exponentially, this overhead quickly becomes insignificant, and the advantage peaks at about 50%. This occurs when $k \approx 15$.

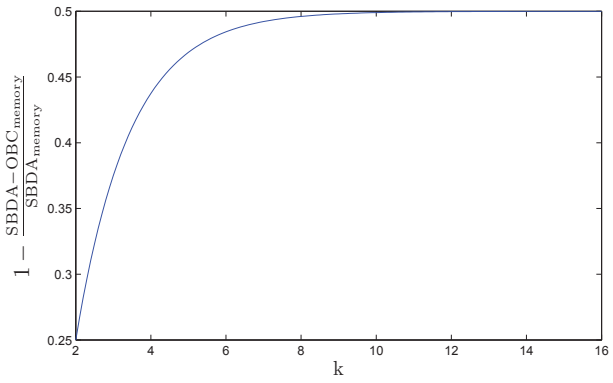

Fig. 5. A plot of the $1 - \frac{SBDA_{memory}}{SBDA\text{-}OBC_{memory}}$ versus k.

Since one of the primary focuses of this paper is the reduction of the computational workload, it would be useful just to focus on only the updating portion. In the following two equations, the number of additions required to update the DA memory table over k samples for SBDA and SBDA-OBC are given below. These equations are for a k-tap sub-filter. Eq. 14 has two terms because the first one, $(k-1)2^{k-2}$, is for updating the table and the second, $(k-1)$, is for computation of the initial condition.

$$SBDA_{adds,updating}(k) = (k-1)(2^{k-1} - 1) \quad (13)$$

$$SBDA - OBC_{adds,updating}(k) = (k-1)2^{k-2} + (k-1) \quad (14)$$

Fig. 6 is a plot of the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for only updating the memory table when k is varied. This advantage is not dependent on K or B_h. It is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks and only provides a benefit when $k > 3$. This figure reaffirms the observation made about Fig. 4 when $k = 2$.

6. CONCLUSIONS

Even in the worst case, the memory usage is still reduced by 25%. This occurs when the size of the sub-filters are small. In most other cases, especially when the sub-filters are large, the memory is reduced by almost 50%. In terms of computational workload, SBDA-OBC is most advantageous for large sub-filters and when the filter is split into few sub-filters. In this case, the computational workload is reduced almost in half.

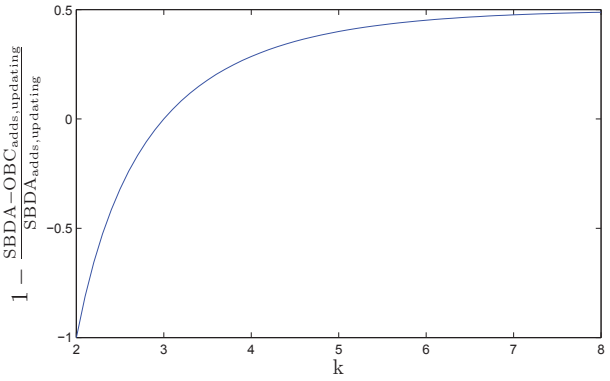

Fig. 6. A plot of the $1 - \frac{SBDA\text{-}OBC_{adds,updating}}{SBDA_{adds,updating}}$ versus k.

7. REFERENCES

[1] D. L. Jones, "Efficient computation of time-varying and adaptive filters," *IEEE Transactions on Signal Processing*, pp. 1077–1086, March 1993.

[2] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, Inc, 1st edition, 1979.

[3] A. Peled and B. Liu, "A new hardware realization of digital filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, pp. 456–462, December 1974.

[4] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, vol. 6, pp. 4–19, July 1989.

[5] C. F. N. Cowan and J. Mavor, "New digital adaptive-filter implementation using distributed-arithmetic techniques," *IEE Proceedings, Part F: Communications, Radar and Signal Processing*, vol. 128, pp. 225–230, August 1981.

[6] C. F. N. Cowan, S. G. Smith, and J. H. Elliott, "A digital adaptive filter using a memory-accumulator architecture: Theory and realization," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 31, pp. 541–549, June 1983.

[7] C.-H. Wei and J.-J. Lou, "Multimemory block structure for implementing a digital adaptive filter using distributed arithmetic," *IEE Proceedings, Part G: Electronic Circuits and Systems*, vol. 133, pp. 19–26, Februray 1986.

[8] D. J. Allred, H. Yoo, V. Krishnan, W. Huang, and D. V. Anderson, "LMS adaptive filters using distributed arithmetic for high throughput," *IEEE Transactions on Circuits and Systems*, vol. 52, no. 7, pp. 1327–1337, July 2005.