PARALLEL COMPUTATION OF ADAPTIVE LATTICE FILTERS

Dong-hwan Lee and Wonyong Sung

School of Electrical Engineering, Seoul National University
Gwanak-gu, Seoul, 151-742 Republic of Korea
Email: ldh@ dsp.snu.ac.kr, wysung@snu.ac.kr

ABSTRACT
Parallel computation of the adaptive lattice filtering algorithm is difficult due to the dependency problem caused by feedback operations. The conventional control-level parallel computation method that exploits the modular structure of the filtering algorithm can only utilize a limited degree of parallelism even when the algorithm is pipeline-transformed. In order to increase the degree of parallelism, we apply the data-level parallel processing method that computes multiple output samples at a time by parallelizing the computation of time-varying linear recursive equations. The control-level parallel processing approach is useful for SIMD (Single Instruction Multiple Data) processor based implementations. However, the data-level parallel processing method is indispensable for multi-core based implementations not only to utilize the increased number of processing cores but also to overcome the communication delay between cores.

Index Terms— Parallel computation, adaptive lattice filter, multi-core CPU, SIMD architecture

1. INTRODUCTION
Adaptive lattice filters are used in many signal processing applications, such as noise cancelation and channel modeling [1]. Adaptive lattice filters not only exhibit quite good convergence characteristics but also have a modular structure that seems advantageous for parallel implementation. There have been several works on VLSI based implementation of adaptive lattice filters, which applies pipelining technique to break the dependency among computation of successive stages [2]. Software based implementations of adaptive lattice filters, which are capable of effectively executing different adaptation algorithms, have also been studied. Meyer and Agrawal suggested a multiple programmable processors based solution that divides the filters into several partitions and assigns each of them to a processor [3]. This method exploits the control-level parallelism for sample by sample processing. A pipelined vector processor based implementation of the adaptive lattice filtering algorithm by using efficient parallel computation method for linear recursive equations was studied [4]. Recently, SIMD (Single Instruction Multiple Data) and multi-core computer architectures are widely adopted for PC and embedded computer systems. However, speeding-up the computation of the adaptive lattice filtering algorithm is not so straightforward especially because of the data-dependency problem. Also, the communication overhead in multi-core systems greatly reduces the efficiency when implementing the adaptive lattice filtering algorithm.

In this research, we develop algorithms and software for parallel implementation of adaptive lattice filters on computer systems that utilize both SIMD and multi-core architecture. Both control- and data-level parallel processing approaches are applied, where the former utilizes the independence of each filter stage while the latter applies the parallel compu-

Fig. 1. Adaptive lattice filter structure. (a) Conventional adaptive lattice filter. (b) Pipelined adaptive lattice filter.
tation method to break the dependency in computation of multiple output samples at a time. The control-level parallel processing approach does not require much overhead but the degree of parallelism is limited to the number of filter stages. The data-level parallel processing approach requires some parallelization overhead as well as extra delay for block processing, although the degree of parallelism supported by this approach is almost unlimited. Both approaches are implemented on a multi-core and SIMD architecture based PC, and their performances are evaluated.

This paper is organized as follows. Section 2 explains parallel computation approaches for adaptive lattice filters. The experimental results are shown in Section 3. Finally, concluding remarks are made in Section 4.

2. PARALLEL COMPUTATION OF ADAPTIVE LATTICE FILTERS

In this section, we focus on the gradient adaptive lattice (GAL) filter proposed by Griffiths (shown in Fig. 1-(a)), and the most widely used version is represented as follows [1, 5].

Initialization:
\[
f_m[0] = b_m[0] = 0, \hat{h}_m[0] = 0, \hat{\kappa}_m[0] = 0, E_{m-1}[0] = a \quad f_0[n] = b_0[n] = u[n] \text{ - input signal} \quad y_{-1}[n] = 0, \|b_{-1}[n]\|^2 = \delta
\]

For filter order \( m \) and time step \( n \):
\[
E_{m-1}[n] = \beta E_{m-1}[n-1] + (1 - \beta) \\
\cdot ([f_{m-1}[n]^2 + b_{m-1}[n-1]^2])
\]
\[
f_m[n] = f_{m-1}[n] + \hat{\kappa}_m[n-1]b_{m-1}[n-1] \quad b_m[n] = b_{m-1}[n-1] + \hat{\kappa}_m[n-1]f_{m-1}[n] \quad (3)
\]
\[
\hat{\kappa}_m[n] = \hat{\kappa}_m[n-1] - \frac{\mu}{E_{m-1}[n]} \quad (4)
\]
\[
y_m[n] = y_{m-1}[n] + \hat{h}_m[n]b_m[n] \quad (5)
\]
\[
e_m[n] = d[n] - y_m[n] \quad (6)
\]
\[
\|b_m[n]\|^2 = \|b_{m-1}[n]\|^2 + |b_m[n]|^2 \quad (7)
\]
\[
\hat{h}_m[n+1] = \hat{\kappa}_m[n] \cdot \hat{\kappa}_m[n-1] \cdot x_k[n] + x_k[n]
\]

Note that the superscript * denotes the complex conjugation, and \( \mu \), \( \delta \), and \( a \) are small positive constants. \( \beta \) is also a constant lying in the range of (0,1). \( M \) is the number of lattice stages, and \( N \) is the total number of input samples.

2.1. Control-level parallel computation

Parallel computation of adaptive lattice filters shown in Fig. 1-(a) is very difficult because of complex dependency relations. One approach to relieve the dependency problem is to pipeline the filter as illustrated in Fig. 1-(b). The inserted pipelining registers isolate the computation of each stage, and the degree of parallelism is equal to the number of filter stages. This parallel computation enables sample by sample processing although the pipelining registers increase the latency between the input and the output. However, the maximum speed-up is limited by the number of lattice stages because more processing units than the number of stages cannot be assigned. Employing multiple cores is not recommended in this method because each core needs to send every intermediate output sample to the next processor, which causes significant communication and synchronization overheads.

2.2. Data-level parallel computation

Another approach is to compute multiple output samples at a time. If we carefully observe the algorithm, the operations correspond to the first order linear recursive equations. Thus, if we employ the parallel computation method of linear recursive equations, even a first order adaptive lattice filter can be implemented with a large degree of parallelism. This parallel computation scheme requires additional memory space that is needed as buffers for input, output, and intermediate variables. Also, the latency of this computation method is very large, and thus this method may not be adequate for real-time control applications. However, the computational procedure does not require fast synchronization between the processing units, and as a result multi-core architecture can be deployed.

2.2.1. Decomposition of adaptive lattice filtering into linear recursive equations

We show that GAL algorithm is composed of multiple linear recursive equations. Equation (1) can be reformulated as follows,
\[
E_{m-1}[n] = \beta E_{m-1}[n-1] + (1 - \beta)(|f_{m-1}[n]|^2 + |b_{m-1}[n-1]|^2)
\]
\[
= \beta E_{m-1}[n-1] + x_k[n].
\]

As shown above, Equation (8) is simply a first order constant coefficient linear recursive equation. Note that \( x_k[n] \) can be computed in parallel because the right hand side variables are already obtained. Equations (2), (3), and (4) look more complicated. To resolve the dependency relation among \( f_m[n] \), \( b_m[n] \), and \( \hat{\kappa}_m[n-1] \), we need to apply equations (2) and (3) into (4).
\[
\hat{\kappa}_m[n] = (1 - |f_{m-1}[n]|^2 + |b_{m-1}[n]|^2) \hat{\kappa}_m[n-1] - 2 \frac{\mu}{E_{m-1}[n]} f_{m-1}[n] b_{m-1}[n] \quad (9)
\]
\[
= a_k[n] \hat{\kappa}_m[n-1] + x_k[n]
\]
The above equation is just a first order time-varying coefficient linear recursive equation, which has \( a_k[n] \) and \( x_k[n] \) as its coefficients and input, respectively. Clearly, if we know the values of \( \hat{a}_{m-1}[n] \) for all \( n \), parallel computation of \( f_m[n] \) and \( b_m[n] \) is a simple task. Equations (5), (6) and (7) form joint-process estimation. If \( y_m[n] \) in (6) is replaced with (5), then it also becomes a first order time-varying coefficient linear recurrence equation.

\[
h_m[n+1] = (1 - \hat{b}_m[n]) \cdot h_m[n] + \frac{\hat{b}_m[n]}{\|b_m[n]\|^2} \cdot (d_m[n] - y_{m-1}[n])
\]

(10)

As shown above, GAL algorithm incorporates three linear recursive equations (one constant and two time-varying coefficient ones) and two non-recursive ones. By using the block processing method, the recurrence ones are efficiently parallelized, which leads to complete parallelization of the GAL algorithm.

2.2.2. Data-level parallel processing of linear recursive equations

A first order linear recursive equation is represented as follows:

\[
y[n] = a[n] \cdot y[n-1] + x[n].
\]

(11)

In this block processing method, multiple blocks are processed simultaneously, employing the data structure organized in an \( L \times P \) array as shown in Fig. 2. The first block has \( x[0], x[1], ..., x[L-1] \), the second block contains \( x[L], x[L+1], ..., x[2L-1] \), and so on. Computing one data block demands for the last output sample of the previous block, which is referred to the initial condition. Once we obtain the initial conditions for all the blocks, there is no dependency among the blocks, thus multiple blocks can be processed at the same time. In the block processing method, the solutions are computed in three stages. First, output samples with zero initial conditions, so called particular solutions, are computed. Second, correct initial conditions are found. At the last, the complete solutions are acquired using the initial conditions [4].

The \( j \)th output samples of \( i \)th data block is computed as follows,

\[
y[iL + j] = a[iL + j] \cdot y[iL + j - 1] + x[iL + j],
\]

(12)

where \( 0 \leq i \leq P - 1 \) and \( 0 \leq j \leq L - 1 \). The above equation can be expanded as follows.

\[
y[iL + j] = a[iL + j] \cdot a[iL + j - 1] \cdots a[iL] \cdot y[iL - 1] + y_p[iL + j] = w[iL + j] \cdot y[iL - 1] + y_p[iL + j]
\]

\[
y[iL + j] = y[iL + j] + y_p[iL + j]
\]

(13a)

where

\[
w[iL + j] = a[iL + j] \cdot a[iL + j - 1] \cdots a[iL] + y_p[iL + j] = w[iL + j] \cdot y[iL - 1] + y_p[iL + j] = y[iL + j] + y_p[iL + j]
\]

(13b)

Particular solution \( (y_p[iL + j]) \) in (13b) can be obtained by evaluating (12) with zero initial condition. Because we assume zero initial condition, the particular solutions of multiple blocks can be computed concurrently. However, these are not the complete solution \( (y[iL + j]) \), thus the transient solution \( (y_t[iL + j]) \), which is a product of the weighting factor \( (w[iL + j]) \) and the initial condition \( (y[iL - 1]) \), must be added. By multiplying coefficients \( (a[iL]) \) through \( (a[iL + j]) \) one by one, the weighting factors can be computed, and there is no inter-block dependency in here. Note that the weighting factors can be computed with particular solutions at the same time. As a result, the data-dependency among the blocks lies only in the initial condition.

The initial condition of \( i \)th block is described as follows,

\[
y[iL - 1] = y[iL - 1] + y_p[iL - 1] = w[iL - 1] \cdot y[(i - 1)L - 1] + y_p[iL - 1].
\]

(14)

The above equation can be sequentially evaluated by a single processing unit because \( w[iL - 1] \) and \( y_p[iL - 1] \) are the results of the previous block. Once correct initial conditions are found, there is no inter-block dependency in computing the transient solutions, and thus complete solutions of multiple blocks can be obtained concurrently.

3. EXPERIMENTAL RESULTS

We conducted experiments at a desktop PC equipping Intel(R) Core(TM) i7-920 (2.66GHz, 4 cores). The CPU has a 64KB L1 cache (32KB instruction, 32KB data) per core, 1MB of total L2 cache, and a 8MB L3 cache that is shared across all the cores. This system supports 4-way SIMD floating point arithmetic with the SSE 4.2 instruction set. To compare the

Fig. 2. Data layout for block processing algorithm.
Table 1. Execution time and speed-up (%) of sequential and parallel implementations of GAL algorithm (L = 64K, time is measured in ms)

<table>
<thead>
<tr>
<th></th>
<th>M = 4</th>
<th>M = 8</th>
<th>M = 12</th>
<th>M = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential(L)</td>
<td>37.68</td>
<td>71.30</td>
<td>105.4</td>
<td>139.7</td>
</tr>
<tr>
<td>Control level parallel. (SIMD, L)</td>
<td>14.79 (255%)</td>
<td>25.14 (284%)</td>
<td>38.59 (273%)</td>
<td>55.30 (253%)</td>
</tr>
<tr>
<td>Data level parallel. (SIMD, L)</td>
<td>13.50 (279%)</td>
<td>23.38 (305%)</td>
<td>33.39 (316%)</td>
<td>43.41 (322%)</td>
</tr>
<tr>
<td>Data level parallel. (multi-core, 4L)</td>
<td>84.09 (179%)</td>
<td>153.0 (186%)</td>
<td>241.4 (175%)</td>
<td>306.9 (182%)</td>
</tr>
<tr>
<td>Data level parallel. (multi-core SIMD, 4L)</td>
<td>44.56 (338%)</td>
<td>66.42 (429%)</td>
<td>99.04 (426%)</td>
<td>128.5 (435%)</td>
</tr>
</tbody>
</table>

Even though GAL algorithm contains complex arithmetics, each of the first order linear recursive equation is a memory bounded problem, and we can expect only limited speed-ups when the block processing is applied. Multi-core based data-level parallel computation shows relatively lower speed-ups than that of SIMD based one despite the fact that both implementations offer the same degree of parallelism. This is because no inter-core communication and synchronization overheads are required in SIMD architecture based data-level parallel computation.

4. CONCLUDING REMARKS

We have implemented gradient adaptive lattice (GAL) filtering algorithms on multi-core and SIMD (Single Instruction Multiple Data) architecture based computer systems. Parallel implementation of the conventional sequential algorithm does not result in considerable speed-up due to data-alignment and communication overheads. The pipelined version of adaptive lattice filters shows good speed-up when employing SIMD instructions, but cannot utilize the multi-core architecture due to large communication and synchronization overheads. In the data-level parallel processing approach, a considerable speed-up can be obtained by utilizing both multi-core and SIMD architecture. These parallel computation methods can be applied to other adaptive lattice filters such as QRD-LSL [1].

5. REFERENCES