Proc. IWSSIP, pages 65-68, Bratislava, June 2008, doi:10.1109/IWSSIP.2008.4604368

Parallelization of IIR Filters Using SIMD Extensions

Rade Kutil
University of Salzburg
Jakob Haringer-street 2, 5020 Salzburg, Austria
Phone: (43) 662 8044/6303, Fax: (43) 662 8044/172, E-mail: rkutil@cosy.sbg.ac.at

Keywords: SIMD, short vector, signal processing, IIR filter

Abstract – The SIMD parallelization of IIR or recursive filters is more difficult than that of FIR filters due to additional data dependencies. While other methods concentrate on appropriate scheduling to enable SIMD parallel execution, this paper proposes a new method where data dependencies are resolved by fusing the recursive application of filter taps into single coefficients. In this way the overhead over perfect parallelity can be reduced to one vector multiply-accumulate operation. Speedups from 1.5 to 4.5 can be obtained with the 4-way SIMD Intel SSE extension, depending on the number of filter taps.

1. Introduction

The parallelization of FIR filters has been investigated thoroughly, especially for wavelet filters, for old SIMD arrays [1], [2], [3] and SIMD extensions of modern general purpose processors in the 1-D case [4], [5] and the 2-D case [6], [7], [8]. The parallelization of IIR filters is more difficult due to data dependencies. There are several approaches including space-time transformations of loop iterations [9], [10] and algebraic transformations [11]. The approach presented in this paper also uses algebraic transformations, although different ones.

From a computational point of view, the difference between FIR and IIR filters lies in the dependencies between loop iterations. Basically, there are two loops, one over signal data and one over filter taps. In the FIR case, iterations of the outer loop, i.e. entire inner loops, are independent of each other, leading to a rather straightforward SIMD parallelization where the two loops (inner and outer) are interchanged for a number of outer iterations equal to the SIMD vector size p. See [4], [5] and Fig. 1(a). In the IIR case, the dependencies are more complicated since all previous output values are required to calculate a new one. See Fig. 1(b). Therefore, SIMD parallelization is more difficult.

This work presents a new approach where data dependencies are resolved by fusing the recursive application of filter taps into single coefficients. In this way it is possible to reduce the overhead of vector operations to only one vector multiply-accumulate operation, apart from several necessary shuffle operations, if the number of filter taps is not smaller than the SIMD parallelity.

All results in this work have been conducted on an Intel Pentium 4 CPU with 3.2GHz and 2MB cache size using the SSE extension with packed words of 4 single precision numbers. All implementations use the same amount of code optimization, i.e. memory access through incremented pointers instead of indexed arrays, and compilation with gcc 4.1.2 with the –O3 option. SIMD operations are implemented using gcc's built-in intrinsics for vector extensions and the –msse option. Note that in order to have full control over generated code, no automatic vectorization is applied.

The results are compared to hand-optimized code by human experts, i.e. the Intel Integrated Performance Primitives (IPP) v5.3, and are able to compete with and outperform it depending on the number of filter taps. Note that the IPP library also uses SIMD operations, but the applied methods are not known to the author.

2. Sequential Algorithm

The goal of IIR filtering is to calculate the signal y from the signal x by

$$y_n = \sum_{i=0}^{N-1} a_i y_{n-i} + \sum_{i=1}^{M-1} b_i x_{n-i}\,,$$

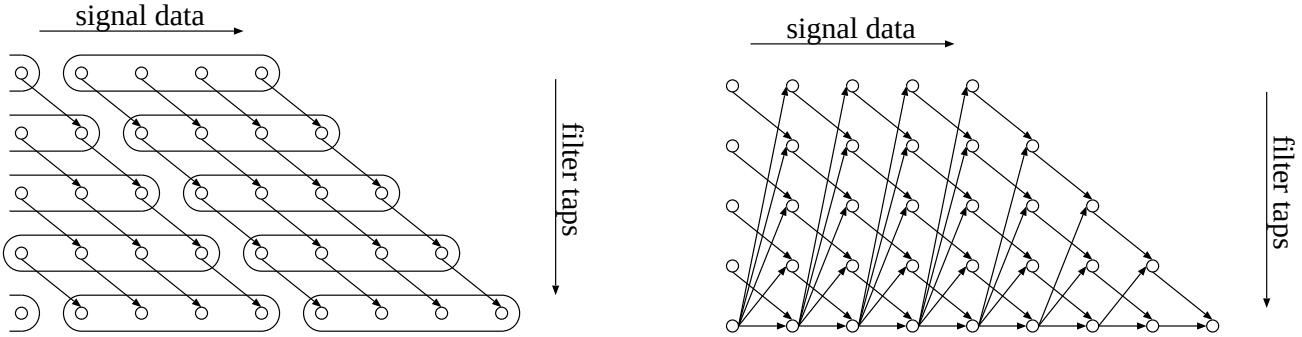
where the second term is an FIR part with coefficients b_i and the first term is the IIR part with coefficients a_i. M is the number of FIR filter taps and N is the number of IIR filter taps. The formula reveals the outer loop over n and two inner loops over i.

The sequential implementation is optimized for performance in order to allow a reasonable comparison to the SIMD parallelized version. It turns out that maintaining a pointer for y_n and x_n and addressing x_{n-i} and y_{n-i} via relative addressing is fastest. Using extra buffers or local register variables for reused values does *not* improve the performance. Therefore, a similar implementation style is adopted for the SIMD parallelization.

3. SIMD parallelization of the FIR part

For SIMD parallelization, it is best when neighboring data has to be processed independently because this leads to a natural sequence of vector operations without the need to combine elements of the same vector, which would involve shuffle operations and incomplete vector operations. The approach to interchange the inner and outer loop for as many outer loop iterations as there are vector elements, leads exactly to this situation and turns out to be near optimal. See Fig. 1(a). The algorithm can be expressed by

$$u = \sum_{i=0}^{M-1} x_{(n-i,\ldots,n-i+p-1)} \odot (b_i, \ldots, b_i)\,,$$



(a) FIR case (b) IIR case

Fig. 1. Loop dependencies in filtering algorithm.

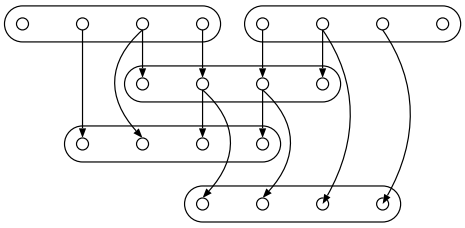


Fig. 2. Shuffle operations for all vector realignments on Intel architecture.

where $\odot$ is the vectorized multiplication and p is the vector size.

However, the input signal x has to be read in a non-aligned way in this approach. This requires a shuffle operation for each non-aligned read. Moreover, there are architectures where not all shuffle operations are possible, e.g. the Intel Pentium architecture. This special architecture demands that the first two of four vector elements originate from the first vector operand and the other two from the second operand. Fig. 2 shows how all possible realignments can nevertheless be implemented on this architecture with one shuffle operation for each non-aligned read.

4. SIMD PARALLELIZATION OF THE IIR PART

The IIR part can be parallelized in just the same way for those iteration where $i \geq p$, i.e. where the source vector $y_{(n-i,\ldots,n-i+p-1)}$ does not overlap with the destination vector $y_{(n,\ldots,n+p-1)}$ that is being calculated. The iterations $i = 0, \ldots, p-1$ might be implemented sequentially after computing the others in a vectorized way first by

$$v = u \oplus \sum_{i=p}^{N-1} y_{(n-i,\ldots,n-i+p-1)} \odot (a_i, \ldots, a_i)\,,$$

followed by

$$y_{n+k} = v_k + \sum_{i=1}^{p-1} a_i y_{n+k-i} \quad \text{for} \quad k = 0, \ldots, p-1.$$

A first attempt to parallelize the latter part is to split it into two phases. The first phase treats those terms that reference y_{n+k-i} where $n+k-i < n$, i.e. values that are already available.

$$\begin{aligned}&\text{for } i = 1, \ldots, p-1 :\\ &\quad v \leftarrow v \oplus (y_{n-p+i}, \ldots, y_{n-1}, 0, \ldots) \odot\\ &\qquad (a_{p-i}, \ldots, a_{p-i}, 0, \ldots)\end{aligned}$$

The second phase uses those elements of v that already represent y_{n+k} values. At the beginning, only $v_0 = y_n$. So, $v_0 a_i$ can be added to v_i for $i = 1, \ldots, p-1$. After that $v_1 = y_n + 1$. Repeating this for $v_2, v_3, \ldots$, leads to the following algorithm:

$$\begin{aligned}&\text{for } k = 0, \ldots, p-2 :\\ &\quad v \leftarrow v \oplus (\ldots, 0, v_k, \ldots, v_k) \odot (\ldots, 0, a_1, \ldots, a_{p-1-k})\\ &y_{(n,\ldots,n+p-1)} \leftarrow v\end{aligned}$$

This first approach yields an overhead of $p-1$ multiply-accumulate vector operations, since each phase has $p-1$ iterations, resulting in $2(p-1)$ operations, where only $p-1$ would be necessary if there were no problems with data dependencies.

Now, we will develop the novel approach that fuses filter taps to resolve data dependencies. Let us look at the second iteration ($k = 1$) of the last algorithm. Here, $v_1 = y_{n+1} = v'_1 + v_0 a_1$, where v' comes from the preceding iteration. Now, we calculate the new v_2 as $v_2 + v_1 a_1$, which can consequently be expressed as $v_2 + v'_1 a_1 + v_0 a_1^2$. Moreover, $v_2 = v'_2 + v_0 a_2$, as calculated in the first iteration. Altogether, we get $v'_1 a_1 + v_0(a_1^2 + a_2)$. The term $v'_1 a_1$ could be calculated in the last iteration of the first phase, and the term $v_0(a_1^2 + a_2)$ can be calculated in the first iteration of the second phase because we have eliminated v_1 from the term.

Following this approach even further recursively, we get the following algorithm that substitutes both phases.

$$\begin{aligned}&\text{for } i = 1, \ldots, p :\\ &\quad v \leftarrow v \oplus (y_{n-p+i}, \ldots, y_{n-1}, 0, v_{p-i}, \ldots, v_{p-i}) \odot s(i)\\ &y_{(n,\ldots,n+p-1)} \leftarrow v\end{aligned}$$

$s(i)$ holds the fused filter tap coefficients and has the

following form:

$$\begin{aligned} s(1) &= (a_{p-1}, \ldots, a_{p-1}, 0) \\ s(2) &= (a_{p-2}, \ldots, a_{p-2}, 0, c_1) \\ &\ldots \\ s(p-1) &= (a_1, 0, c_1, c_2, \ldots, c_{p-2}) \\ s(p) &= (0, c_1, c_2, \ldots, c_{p-1})\,, \end{aligned}$$

where

$$c_k = \sum_{i=1}^{k} a_k c_{k-i}, \qquad c_0 = 1\,.$$

This approach finally has an overhead of only one multiply-accumulate vector operation, since it has p iterations. For better comprehensibility, let us write the algorithm or the case of $p = 4$ as in the Intel SSE architecture:

$$\begin{aligned} v &\leftarrow v \oplus (y_{n-3}, y_{n-2}, y_{n-1}, 0) \odot (a_3, a_3, a_3, 0) \\ v &\leftarrow v \oplus (y_{n-2}, y_{n-1}, 0, v_2) \odot (a_2, a_2, 0, a_1) \\ v &\leftarrow v \oplus (y_{n-1}, 0, v_1, v_1) \odot (a_1, 0, a_1, a_1^2 + a_2) \\ v &\leftarrow v \oplus (0, v_0, v_0, v_0) \odot (0, a_1, a_1^2 + a_2, a_1^3 + 2a_1a_2 + a_3) \\ y_{(n,\ldots,n+3)} &\leftarrow v \end{aligned}$$

Of course, each of these multiply-accumulate operations requires at least one shuffle operation, maybe two on the Intel SSE architecture.

If the number of IIR-taps N is smaller than the vector size p, the above approach unfortunately only reduces to $p - 1$ operations. In this case, some divide-and-conquer algorithm might further reduce the overhead. However, $\lceil \log_2(p+1) \rceil$ seems to be the lower bound, since y_{n+p-1} depends on the $p+1$ values $u_0, \ldots, u_{p-1}, y_{n-1}$ if N takes the minimal value 2.

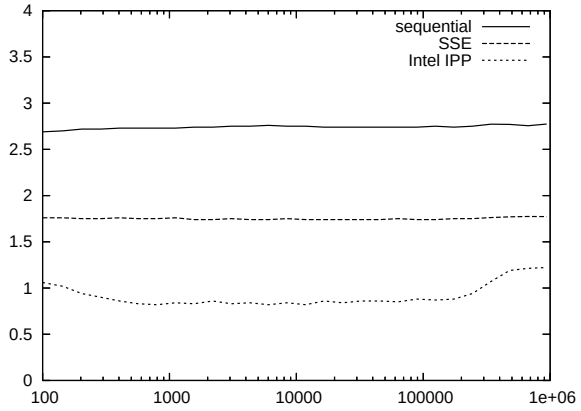
5. Performance

In [4], [5], it turned out that the performance of an implementation of a filtering algorithm possibly depends on whether the signal data is in the cache or not. The method to find this out is to vary the data length and to repeat the filtering several times. If the data is short, then it will remain in the cache, otherwise it will not. This approach is also adopted here.

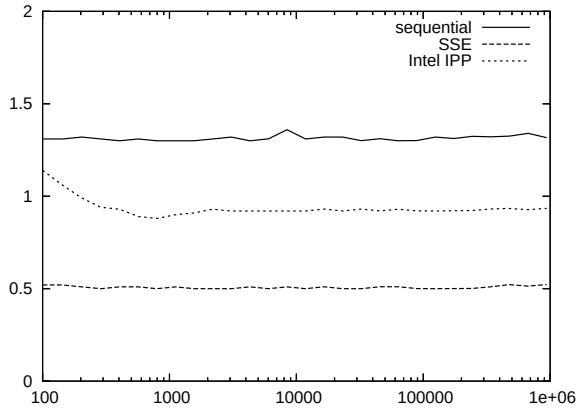
The calculation time is expected to depend linearly on the data size and on the number of filter taps $N + M$. Therefore, we calculate the execution time per sample point and filter tap from the total execution time of the algorithm by $t_{\text{total}}/S/(N+M)$, where S is the data size.

Fig. 3 shows the results for $N = M = 2$ and $N = M = 10$. It also includes performance measures of the Intel IPP library. While the IPP library code seems to depend a little on the data size, the major reason for this seems to be startup-overhead when filling the delay-lines, which is significant only for small data sizes. The sequential algorithm and the SIMD algorithm are completely independent of the cache state.

For small numbers of taps, the IPP library code seems to be faster. This is also shown in Fig. 4. For $N = M \leq 5$, the SIMD algorithm cannot compete with the IPP code. The



(a) 2 taps



(b) 10 taps

Fig. 3. Execution time in ns per sample point and filter tap depending on the data length for repeated filtering, showing the cache dependency of the algorithms.

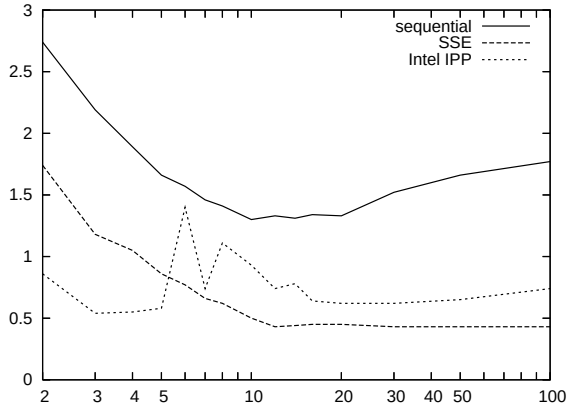


Fig. 4. Execution time in ns per sample point and filter tap depending on the number of filter taps.

reason is probably that hand-optimizing assembler code, as in the IPP library, is more important for short loops. For $N > 5$, however, the new SIMD approach outperforms the IPP library by a speedup of about 1.7 and also shows more regular behavior. Compared to the sequential algorithm, speedups from 1.5 for small N to 4.5 for large N are obtained.

6. CONCLUSION

A successful approach for FIR filtering on SIMD architectures is based on the interchange of the inner loop over filter taps and the outer loop over the signal data. If this is done on a number of outer iterations that is equal to the vector size p, a convenient SIMD algorithm can be generated easily.

In the IIR case, additional data dependencies disturb this scheme. However, it has been shown that the scheme can be left unchanged for all but the first p filter taps if the order of filter tap application is reversed. For those taps, coefficients can be fused in order to resolve data dependencies. This algebraic transformation manages to limit the parallel overhead to one vector multiply-accumulate operation.

The approach yields speedups of 1.5 to 4.5 compared to the sequential version on an Intel Pentium SSE processor. The fact that the speedups exceed $p = 4$ is probably due to improved cache usage. It even outperforms the hand-optimized code of the Intel IPP library by a speedup of about 1.7 if the number of filter taps is 6 or higher.

ACKNOWLEDGMENTS

The author wants to thank Robert Resch, Wolfgang Kreil and Armin Langhofer for co-developing and testing first special versions of the approach, showing that the approach is feasible.

REFERENCES

[1] M.M. Pic, H. Essafi, and D. Juvin. Wavelet transform on parallel SIMD architectures. In F.O. Huck and R.D. Juday, editors, *Visual Information Processing II*, volume 1961 of *SPIE Proceedings*, pages 316–323. SPIE, August 1993.

[2] C. Chakrabarti and M. Vishvanath. Efficient realizations of the discrete and continuous wavelet transforms: From single chip implementations to mappings on SIMD array computers. *IEEE Transactions on Signal Processing*, 3(43):759–771, 1995.

[3] M. Feil and A. Uhl. Wavelet packet decomposition and best basis selection on massively parallel SIMD arrays. In *Proceedings of the International Conference "Wavelets and Multiscale Methods" (IWC'98), Tangier, 1998*. INRIA, Rocquencourt, April 1998. 4 pages.

[4] R. Kutil, P. Eder, and M. Watzl. SIMD parallelization of common wavelet filters. In *Parallel Numerics '05*, pages 141–149, Portorož, Slovenia, April 2005.

[5] R. Kutil and P. Eder. Parallelization of wavelet filters using SIMD extensions. *Parallel Processing Letters*, 16(3):335–349, September 2006.

[6] R. Kutil. A single-loop approach to SIMD parallelization of 2-D wavelet lifting. In *Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 413–420, Montbeliard-Sochaux, France, February 2006.

[7] C. Tenllado, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2D wavelet lifting transform using SIMD extensions. In *Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia, PDIVM '03*, Nice, France, April 2003.

[8] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2-D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *Proceedings of the 2000 International Conference on High Performance Computing*, Bangalore, India, December 2002.

[9] R. Schaffer, M. Hosemann, R. Merker, and G. Fettweis. Recursive filtering on SIMD architectures. In *Proceedings of the IEEE Workshop on Signal Processing Systems 2003 (SIPS 2003)*, pages 263–268, August 2003.

[10] M. Hosemann and G. Fettweis. On enhancing SIMD-controlled DSPs for performing recursive filtering. *Journal of VLSI signal processing*, 43(2–3):125–142, June 2006.

[11] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. Implementation of recursive digital filters into vector SIMD DSP architectures. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing 2004 (ICASSP '04)*, volume 5, pages 165–168, May 2004.