PERFORMANCE ANALYSIS OF ALTERNATIVE STRUCTURES FOR 16-BIT INTEGER FIR FILTER IMPLEMENTED ON ALTIVEC SIMD PROCESSING UNIT

Grzegorz Kraszewski

Białystok Technical University, Department of Electric Engineering
Wiejska 45D, 15-351 Białystok, Poland, e-mail: krashan@teleinfo.pb.edu.pl

Abstract: *The paper presents paralellized structures of 16-bit integer, one-dimensional FIR filters optimized for AltiVec SIMD processing unit used in PowerPC processor family. As FIR filtering, like most of DSP kernels, is memory bandwidth limited, proposed structures minimize number of memory accesses, increasing filter computation speed. Performance of three alternative filter structures is compared and analysed.*

1 Introduction

Finite impulse response digital filter is one of the basic DSP operations, used not only standalone, but being a base for many other algorithms. Mathematically the filter is a convolution operation, where filter coefficients are convoluted with the filtered signal, according to the well known formula:

$$y[k]=\sum_{n=0}^{N-1} x[k-n]\cdot f[n] \quad (1)$$

where $x[n]$ is the filtered signal, and $f[n]$ is the table of filter coefficients containing N elements. For effective SIMD implementation, convolution operation may be treated as multiplication of a matrix made of shifted fragments of input vector, by vector of filter coefficients.

Most of audio applications process 16-bit signed integer samples. That is why I've choosen a 16-bit filter. Accumulators have to be longer however, for high quality processing 32-bit accumulators are needed (assuming filter coefficients are normalized:

$$\sum_{n=0}^{N-1} |f[n]| < 2^{16} \quad (2)$$

it is guarranted no overflow will occur for any input signal).

In the practical implementation input vector is very long, often it can be considered unlimited (for example while processing stream of Internet radio). Filter table has the known length, but I assume here it does not fit into the SIMD register file (filters from 64 to 8192 taps are tested). This approach differs from the one taken by most of researchers [2][4][5][6], but matches practical applications better (FIR of less than 64 taps is not selective enough for many applications).

A general FIR filter cannot be optimized in terms of multiplications and additions. All optimizations of FIR filters are based either on specific properties of filter coefficients (table symmetry for linear phase filters [13], table sparsity for FRM filters [14]), or specific properties of the signal (polyphase filters for resamplers [8]). I do not assume any of those properties in the paper, so number of arithmetic operations does not change compared to the formula (1).

Speedup gained on using SIMD unit comes from two sources. The first one is obviously SIMD parallelism of computation, as single instruction processes 8 samples at once. The second is reordering of calculations in a way which reduces number of memory reads per one signal sample.

Implementation of FIR filters on different SIMD units were a subject of some previous papers. Unfortunately most of them investigate only very short filters (usually below 64 taps), and doesn't take into account limited memory bandwitdh. Intel application note for MMX [9] describes 16-tap filter, application note from Motorola [11] describes a few short filters up to 64 taps. In [2] author does not specify tested FIR parameters, he only mentions the dataset fits L1 cache (which is a bit unrealistic, especially for filter input vector). In the work [4], a 32-tap filter is ran in a loop over the same 256 samples of input. Similarly the paper [5] presents 35-tap FIR running in a loop over set of 4000 samples.

FIR filter architectures presented in the paper are implemented as a class in *Reggae* [16], an object oriented streaming multimedia processing framework for *MorphOS* operating system. That is why working on extremely long data streams and taking into account the limited memory bandwidth of the hardware are principles of proposed designs.

2 Calculations reordering

Reordering of FIR calculation is critical for the algorithm pefromance if memory bandwidth is limited. It has been shown, that most of simple DSP operations, like FIR filters are memory bound [1][2][3]. In typical conditions SIMD unit calculating a convolution is at

least a few times faster than memory bus. For *Pegasos II* machine used for measurements *vec_msum()* instruction performed in a tight, unrolled loop needs data at a rate of 32 GB/s, while main memory controller can deliver data at only 225 MB/s (two-level cache helps a lot, but does not solve the problem).

As mentioned earlier, digital convolution may be dentoed as matrix by vector multiplication. To avoid negative indexes, N-1 zeros are added at the start of input data, then filter table is mirrored, so both signal and filter table indexes can be incremented in the loop. After these simple operations, FIR filter can be denoted as following multiplication:

$$Y = X \cdot F = \begin{bmatrix} x[0] & x[1] & x[2] & \dots & x[N-1] \\ x[1] & x[2] & x[3] & \dots & x[N] \\ x[2] & x[3] & x[4] & \dots & x[N+1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix} \cdot \begin{bmatrix} f[0] \\ f[1] \\ \vdots \\ f[N-1] \end{bmatrix} \quad (3)$$

where Y is output vector, F is a mirrored filter coefficients table and the matrix X is made of shifted copies of input preceeded with N-1 zeros. Straightforward implementation performs this multiplication row by row, which requires $2N$ memory reads for every output sample. Taking advantage of the large (32 × 128 bit) AltiVec register file, calculation can be reordered, then many rows are multiplied in one pass. It dramatically reduces number of memory reads. For example if 8 rows (and 8 output samples) are calculated in one pass (it will be called 8-pipe algorithm, in the paper), every filter coefficient is used 8 times once loaded into a register, the same for most of input samples. Formally for m-pipe algorithm for filter of N taps we need to load N filter coefficients and N+m-1 input samples for m output samples. Then number of memory reads per sample can be then given as:

$$R_{PS} = \frac{2N + m - 1}{m} \quad (4)$$

3 Available SIMD instructions

There are different AltiVec instructions that can be used for integer 16-bit FIR with 32-bit accumulation [12] [10]. The most promising one is *vec_msum()*, which performs eight 16-bit by 16-bit multiplications with 32-bit product and 8 32-bit additions. An alternative set of instructions would be *vec_mule()* and *vec_mulo()* accompanied by *vec_add()*. Set of these three instructions has lower number of operations per single processor instruction however, so only filter architectures based on *vec_msum()* will be analysed. Because of different sizes of arguments (16-bit) and results (32-bit) the instruction uses a bit unusual layout of arguments (Fig. 1).

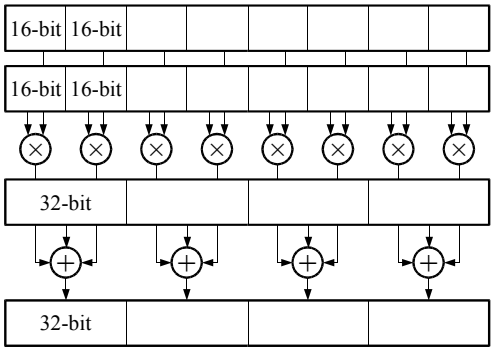

Fig. 1. Arguments layout for *vec_msum()* instruction.

4 Three alternative structures

Because of *vec_msum()* arguments layout some permutation operations for arguments ordering are unavoidable. There are three variants possible:

1. Input data are fetched without permutation, output data have to be summed across register, permuted, then reduced to 16-bit. (architecture A).

2. Input data are permuted, so output is in-order, it has to be reduced to 16-bit however (architecture B).

3. Some permutation is done on input, some on output, no across register summation (architecture C).

For all three architectures most of matrix X rows must be produced by data shifting. As every AltiVec register contains 8 samples, only every 8-th row may be loaded directly, the rest is generated with *vec_sld()* instruction, which is byte-precise left shift across two registers.

4.1 Architecture A

In this architecture input data are not permuted, so one *vec_msum()* argument consists of 8 consecutive input samples, the second one is made of 8 consecutive filter coefficients, so one *vec_msum()* works for one output sample. An output sample is accumulated across an AltiVec register in four 32-bit partial results. A disadvantage of architecture A is that every output sample requires the whole 128-bit register. It limits possible parallelizing to 16-pipe filter (I assume number of pipes is a power of two, 16-pipe filter uses 24 AltiVec registers out of 32 available). On the other hand using 4 partial results allow for gaining additional 2 bits for accumulation, as one partial result sums only ¼ of multiplications. It is important for very long filters, where normalization requirement (2) limits dynamic range of coefficients. If these additional 2 bits are used, partial results are shifted 2 bits right before across-register summation done with *vec_sums()*. After that, 8 sums are truncated to 16 bits by discarding lower half, permuted into one register and stored. The figure 2

shows an example of computation order for $N = 16$ and $m = 8$ on the matrix X in (3). One small rectangle represents two elements of the matrix.

1	1	1	1	9	9	9	9
2	2	2	2	10	10	10	10
3	3	3	3	11	11	11	11
4	4	4	4	12	12	12	12
5	5	5	5	13	13	13	13
6	6	6	6	14	14	14	14
7	7	7	7	15	15	15	15
8	8	8	8	16	16	16	16

Fig. 2. Order of computations for the FIR architecture A on matrix X with $N = 16$ and $m = 8$.

4.2 Architecture B

The principle of this design is to perform all permutations on input data. The order of computation of matrix X is shown on the figure 3. In this case one *vec_msum()* instruction works for 4 output samples. Note that every input sample is repeated twice before passed to *vec_msum()*. For example an argument for the first *vec_msum()* must be:

$$\begin{bmatrix} x[0] & x[1] & x[1] & x[2] & x[2] & x[3] & x[3] & x[4] \end{bmatrix} \quad (5)$$

It can't be done with shifting, so every time it must be done with *vec_perm()*. Filter coefficients have to be permuted as well, *vec_split()* repeats chosen two across a register. The advantage of architecture B is that an output register holds 4 accumulators instead of one as in architecture A. This saves some registers and allows for 32-pipe computations. The disadvantage is a high number of permutations.

1	3	5	7	9	11	13	15
1	3	5	7	9	11	13	15
1	3	5	7	9	11	13	15
1	3	5	7	9	11	13	15
2	4	6	8	10	12	14	16
2	4	6	8	10	12	14	16
2	4	6	8	10	12	14	16
2	4	6	8	10	12	14	16

Fig. 3. Order of computations for the FIR architecture B on matrix X with $N = 16$ and $m = 8$.

4.3 Architecture C

The third proposed architecture is some compromise between previous two. Accumulator registers are interleaved, so half of them holds only even samples of the output, the second half holds odd samples (for 8-pipe filter one register holds just samples 0, 2, 4, 6, the second one holds 1, 3, 5, 7). Then input vectors passed to *vec_msum()* can be produced by simple shifts as in the architecture A, but one *vec_msum()* works for 4 output samples (and one accumulator register holds 4 output samples) as in the architecture B, so 32-pipe routine can be used. Interleaving output samples and discarding lower 16 bits can be done in the same permutation. The figure 4 shows computation order for this architecture.

1	3	5	7	9	11	13	15
2	4	6	8	10	12	14	16
1	3	5	7	9	11	13	15
2	4	6	8	10	12	14	16
1	3	5	7	9	11	13	15
2	4	6	8	10	12	14	16
1	3	5	7	9	11	13	15
2	4	6	8	10	12	14	16

Fig. 4. Order of computations for the FIR architecture C on matrix X with $N = 16$ and $m = 8$.

5 Implementations and their performance

The three described architectures of FIR filter have been implemented on *Pegasos II* machine equipped with *PowerPC 7447 (G4)* processor clocked at 1.0 GHz, and using DDR-266 memory on 133 MHz FSB bus. The processor has 32 kB of L1 cache and 512 kB of L2 cache. Measured main memory access speed is 225 MB/s for reading and 470 MB/s for writing. Filter routines and benchmarking program have been written in C and compiled with GCC 2.95.4 compiler. The source code can be found in [15]. Except of three SIMD implementations, two other routines have been measured, a straightforward, "naive" implementation of the formula (1), and memory access optimized scalar (not using SIMD unit) 8-pipe routine taking advantage of 32 GPR registers in the processor.

For the three AltiVec architectures numbers of executed AltiVec instructions were counted (by analyzing disassembled executable), for 1024-tap filter ($N = 1024$) and 1 048 576 (2^{20}) input samples. Results are presented in the table below.

Tab. 1. Instruction count for three FIR architectures.

Instruction	Arch. A 16-pipe	Arch. B 32-pipe	Arch C. 32-pipe	Unit
vsplitb	0	2	0	VPU
vsplitw	1	32 768	32 768	VPU
vspltw	0	16 777 216	16 777 216	VPU
vlsdoi	118 095 873	8 552 448	121 798 656	VPU
vperm	1 048 576	134 348 800	131 072	VPU
lvx	16 908 288	8 519 680	8 519 691	LSU
stvx	131 072	131 072	131 072	LSU
vor	17 170 432	8 454 144	12 648 448	VIU1
vaddubm	0	16 777 216	0	VIU1
vsububm	0	12 582 916	0	VIU1
vmsumshm	134 217 728	134 217 728	134 217 728	VIU2
vsumsws	1 048 576	0	0	VIU2

As it can be seen in the table, number of MAC instructions (*vmsumshm*) and store instructions (*stvx*) is the same in all three architectures. It should be, because

proposed three architectures do not change number of multiplications and additions, store instructions are used only to store output samples. What is worth noting, is the impact of multi-pipelining on load (*lvx*) instructions. Without multi-pipelining there will be $2.147 \cdot 10^9$ 16-bit loads (according to the formula (2)), which means $2.684 \cdot 10^8$ *lvx* loads (128-bit). When *N* in the formula (2) is large, having *m* pipelines instead of one reduces number of memory loads *m* times.

Real performance of described filters has been measured with a benchmark program (which source code is available in [15]). Benchmark measures performance in Mtaps/s (millions filter taps per second) for different filter lengths from 64 to 8192 taps. The benchmark uses input signal vector of 9 000 000 samples. Performance of filters is presented in the table below.

Tab. 2. FIR filters performance in Mtaps/s for different filter lengths.

Architecture	64	128	256	512	1024	2048	4096	8192
scalar, 1-pipe	229	237	242	245	246	247	247	246
scalar, 8-pipe	361	391	409	418	424	428	429	429
arch. A, 16-pipe	2840	4210	5185	5810	6170	6400	6520	6540
arch. B, 32-pipe	2600	3260	3625	3830	3970	4040	4085	4090
arch. C, 32-pipe	2780	3460	3900	4120	4320	4360	4425	4440

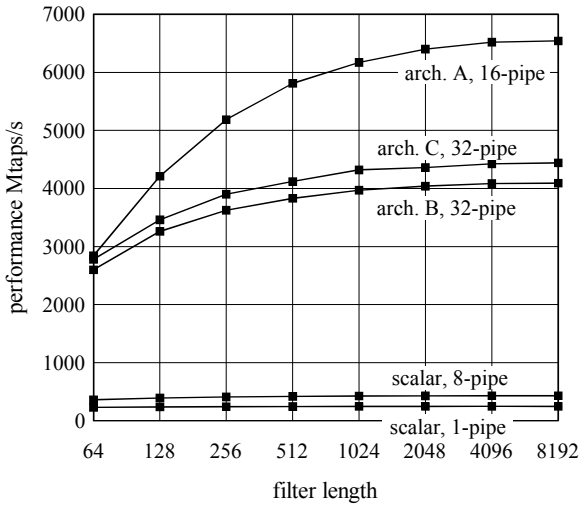

Fig. 5. Performance comparison of different filter architectures.

In spite of only 16 pipes, architecture A outperforms the rest. It is a sign, that memory bound task turns into computationally bound one. The main problem of architectures B and C seem to be overloading Vector Permute Unit of the CPU. In the table below instruction loads for VPU (Vector Permute Unit), VIU1 (Simple Vector Integer Unit), VIU2 (Complex Vector Integer Unit) and LSU (Load and Store Unit) are calculated.

Tab. 3. Instruction load on execution units.

Architecture	VPU	VIU1	VIU2	LSU
arch. A, 16-pipe	$1.19 \cdot 10^8$	$0.34 \cdot 10^8$	$1.34 \cdot 10^8$	$0.16 \cdot 10^8$
arch. B, 32-pipe	$1.60 \cdot 10^8$	$0.38 \cdot 10^8$	$1.34 \cdot 10^8$	$0.08 \cdot 10^8$
arch. C, 32-pipe	$1.39 \cdot 10^8$	$0.12 \cdot 10^8$	$1.34 \cdot 10^8$	$0.08 \cdot 10^8$

The main difference here is the load of VPU unit, which is the lowest for the architecture A.

6 Conclusion

SIMD units in spite of their almost 10-year history and presence in every personal computer, are still a challenge for programmers. Classic optimization techniques as well as autovectorization in compilers, concentrate on register-level parallelization and reduction of arithmetic operations, while many DSP processing tasks are memory bound. Careful optimization prioritized on reducing load/store operations can give enormous speed-up of DSP algorithms implemented on SIMD units and remove memory throughput limits.

The best of proposed FIR routines achieves 80% of theoretical AltiVec unit throughput (7447 unit clocked at 1.0 GHz is able to perform $8 \cdot 10^9$ 16-bit MAC operations with 32-bit accumulation, assuming ideal pipelining and register interleaving) in spite of very limited bandwidth of main memory controller. Speedup over plain scalar code (26.5 times faster) is much more than expected from 8-way SIMD parallelism and shows how important is temporal data locality while designing DSP algorithms for SIMD processing units.

References

[1] Sebot J., Drach-Temam N., *Memory Bandwidth: The True Bottleneck of SIMD Multimedia Performance of Superscalar Processor*, Lecture Notes in Computer Science, vol. 2150/2001, 437.

[2] Sebot J., *A Performance Evaluation of Multimedia Kernels Using AltiVec Streaming SIMD Extensions*, Sixth International Symposium on High Performance Computer Architecture, Toulouse, 2000.

[3] Talla D., John L. K., Burger D., *Bottlenecks in Multimedia Processing with SIMD Style Extensions And Architectural Enchancements*, IEEE Transactions on Computers, Vol. 52, Issue 8, Aug. 2003, 1015–1031.

[4] Talla D., John L. K., Lapinski V., Evans B. L., *Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures*, 2000 IEEE International Conference on Computer Design (ICCD'00), 163.

[5] Nguyen H., John L. K., *Exploiting SIMD Parallelism In DSP And Multimedia Algorithms Using the AltiVec Technology*, Proceedings of the 13th International Conference on Supercomputing, 1999, 11–20.

[6] Bhargava R., John L. K., Evans B. L., Radhakrishnan R., *Evaluating MMX Technology Using DSP And Multimedia Applications*, Proceedings of. 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, 37–46.
[7] Fridman J., *Data Alignment for Sub-Word Parallelism in DSP*, 1999 IEEE Workshop on Signal Processing Systems, 251–260.
[8] Crochiere R. E., Rabiner L. R., *Multirate Digital Signal Processing*, 76–91.
[9] [—], *32-bit Floating Point Real and Complex 16-tap FIR Filter Implemented Using Streaming SIMD Extensions*, Intel 1999.
[10] [—], *MPC7450 RISC Microprocessor Family Reference Manual*, Freescale Semiconductor 2005.
[11] [—], *AltiVec Real FIR [application note]*, Motorola 1998.
[12] [—], *AltiVec Technology Programming Interface Manual,* Motorola 1999.
[13] Lyons R. G., *Wprowadzenie do cyfrowego przetwarzania sygnałów [Understanding Digital Signal Processing]*, 1997, 398–399.
[14] Lim Y. C., *Frequency-Response Masking Approach for the Synthesis of Sharp Linear Phase Digital Filters*, IEEE Transactions on Circuits and Systems, vol. 33, 1986, 357–364.
[15] Kraszewski G., *Source code for AltiVec optimized 16-bit integer FIR filter and benchmark program*, http:// teleinfo.pb.edu.pl/~krashan/altivec/fir16/.
[16] Kraszewski G., *Reggae, the streaming media library for MorphOS*, http://teleinfo.pb.edu.pl/reggae.