Key words:
Digital Signal Processing, FIR filters,
SIMD processors, AltiVec.

Grzegorz KRASZEWSKI
<krashan@teleinfo.pb.edu.pl>
Białystok Technical University
Department of Electrical Engineering
Wiejska 45D, 15-351 Białystok

FAST FIR FILTERS FOR SIMD PROCESSORS WITH LIMITED MEMORY BANDWIDTH

FIR filtering is one of the most populars DSP algorithms. Finite impulse response filters are easy to design, unconditionally stable, phase linearity is easily obtained. Also noise and rounding error analysis are easy. On the other hand FIR filters require high filter order when compared to IIR ones. It means increased number of multiplicatons and additions, and – what is also important – number of memory reads. Current processors and especially their SIMD units like *AltiVec* or *SSE* can perform FIR calculation much faster than memory controller can supply the data. Then reordering of calculations without changing the amount of them, gives significant algorithm speedup. In the paper the idea of FIR reordering is presented, followed by experimental results obtained on *PowerPC G4* processor with *AltiVec* SIMD unit.

1. FIR FILTERS ON SIMD PROCESSORS

FIR filtering, or digital convolution is a common operation in DSP area. Because of its simplicity it can be easily implemented on SIMD type of processors. Parallelization itself, while it is easily applied to FIR filters, is not enough to squeeze the maximum performance from a processor. The classic definition of computation complexity takes into account only number of arithmetic operations, usually in terms of additions and multiplication. What it doesnt't take is an effort needed to fetch data from memory. Current general purpose processors can crunch numbers much faster than fetch them from memory. This reduces the algorithm performance, especially when number of computations per kB of processed data is relatively low (which is the case for FIR filters) [1].

I assume in the paper, a FIR filter is long (has about 50 taps at least) and operates on a long data stream with unknown length. It is the case when processing audio streams for example. The filter is implemented with a well known formula of digital convolution:

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

where $x[n]$ is the input signal and $f[n]$ is the table of filter coefficients. N is the filter order or number of taps. The general filter form cannot be optimized in terms of number of additions and multiplications. There are many optimizations based either on assumptions on input signal (for example polyphase filters [6]), or assumptions on filter coefficients (for example linear phase filters have symmetrical table, which allows for saving of $N/2$ multiplications [11]. Frequency response masking filters have many zero coefficients [12]). The general filter form will be assumed later in the paper.

An N-order FIR filter requires N multiplications and N additions for every output signal sample. It also reqires $2N$ memory read operations, N of them is for input signal, the rest for them is for filter coefficients. As nowadays SIMD processors are very fast, memory reading becomes the bottleneck of the algorithm [1]. For example the machine used for experiments is *Pegasos II* with *PowerPC* 7447 processor clocked at 1.0 GHz. Theoretical maximum memory bandwidth required for vec_fmadd() operator used in FIR implementation is 12 GB/s (assuming no pipeline flushes), while maximum theoretical bandwidth of DDR-266 memory used in the hardware is 2.1 GB/s. Of course two levels of cache memory help much, but load instruction always adds latency, even if data are fetched from L1 cache [8].

Vectorization of FIR filters was the subject of some previous publications. There are application notes by Intel for MMX [7] and by Motorola for AltiVec [9] with accelerated FIR filter code discussions. Only very short filters are discussed however, where the whole filter table fits easily into SIMD register file. In [5] there is a short 35-taps filter investigated with short signals of 4000 samples. Publication [2] mentions some FIR filter being tested, but without specifying any of its parameters. In the paper [4] there is 32-tap filter investigated working repeatedly on the same 256 samples (which fits the whole "data stream" into L1 cache). In my work I've assumed unlimited stream of input signal (I've benchmarked my filters with uncompressed music fragments of 10 to 25 millions of samples). I've also assumed long filters (lenghts from 64 to 8192 taps were tested). My benchmarks have been ran in a real environment of working operating system, which has significant impact on cache availability.

2. REORDERING OF COMPUTATIONS

While one output sample of FIR filter requires $2N$ memory reads, it should be noted, that most of them are loading the same values (signal samples and filter coefficents) as used for the previous output sample (only one sample of input signal is new). It is obvious looking at hardware implementations, where input samples move along a shift register.

Convolution operation may be expressed in terms of vectors and matrices. Then to obtain K samples of output signal, a $K \times N$ matrix composed of shifted copies of input signal is multiplied by a vector containing filter coefficents:

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{K-1} \end{bmatrix} = \begin{bmatrix} x_0 & x_{-1} & x_{-2} & \dots & x_{-N+1} \\ x_1 & x_0 & x_{-1} & \dots & x_{-N+2} \\ x_2 & x_1 & x_0 & \dots & x_{-N+3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{K-1} & x_{K-2} & x_{K-3} & \dots & x_{K-N} \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} \tag{2}$$

To simplify an implementation vector X' is created from input vector X by adding N-1 zero samples in front of X. Another implementational trick is to flip the filter vector F, so indexes in both input vector and filter vector are going upward. After applying both the changes, equation (2) takes following form:

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{K-1} \end{bmatrix} = \begin{bmatrix} x'_0 & x'_1 & x'_2 & \dots & x'_{N-1} \\ x'_1 & x'_2 & x'_3 & \dots & x'_N \\ x'_2 & x'_3 & x'_4 & \dots & x'_{N+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x'_{K-1} & x'_K & x'_{K+1} & \dots & x'_{K+N-2} \end{bmatrix} \cdot \begin{bmatrix} f'_0 \\ f'_1 \\ f'_2 \\ \vdots \\ f'_{N-1} \end{bmatrix} \tag{3}$$

which can be directly implemented with following C code:

```
void fir(float *X, float *F, float *Y, int K, int N)
{
  int k, n;
  for (k = 0; k < K; k++)
  {
    float s = 0.0;
    for (n = 0; n < N; n++) s += F[n] * X[k + n];
    Y[k] = s;
  }
}
```

The preformance of this code will be used as reference level for comparision with AltiVec optimized versions. Optimization gain comes from two sources: the first is four-way parallelism of AltiVec floating point operations, we may expect the code to be four times faster. The second, less obvious source of speedup is increasing data locality by operation reordering. Reference code traverses the matrix X horizontally, row by row. This requires $2N$ reads from memory (be it L1 cache, L2 cache or main memory). Data locality may be increased significantly by dividing the X matrix into rectangular blocks and splitting F and Y vectors accordingly.

The block size will be a multiply of 4, as one AltiVec register can hold 4 floating point numbers. Hence the minimal block is 4×4. Of course we can (and should) use bigger blocks, under the condition, that data needed to compute through a single block have to fit into AltiVec register file (128 numbers). For $p \times q$ block we need $p/4$ registers for filter coefficent, $q/4$ registers for output and $p/4 + q/4$ registers for (shifted) input, the sum is $(p+q)/2$ (4). At the same time number of memory reads (in 32-bit words) per one output sample can be calculated as $(2N + q)/p$ (5).

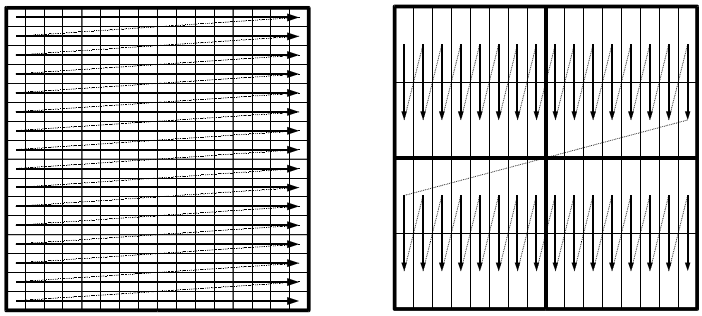


Fig. 1. Order of MAC-s (multiply and accumulate) for 16 output samples and 16-tap filter: reference code (left) and AltiVec code with 8×8 blocks (right).

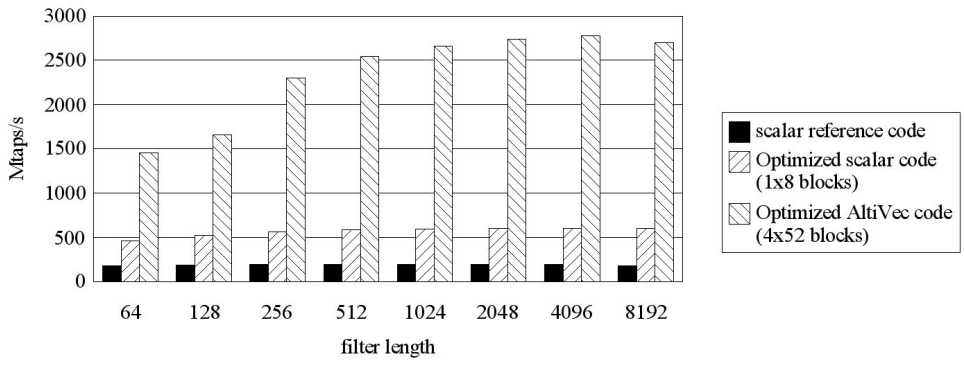
The figure above shows the order of computations for plain reference C code and for AltiVec version using 8×8 blocks. Note that inside a block computation order is transposed – columns are calculated instead of rows. Then a single 4-element MAC operation uses one filter coefficent for all 4 signal samples (and it is reused then for the whole column). It requires additional vec_splat_u32() operation, but has the advantage over row calculation, that accumulation register holds ready output samples, when horizontal scan across the matrix is finished. There is no need for across-register summation, which requires additional output sample gathering before storing in memory.

3. OPTIMAL BLOCK SIZE

We have two constraints for the block size. The first one is number of available AltiVec registers and formula (4). The second one is number of memory reads per output sample (5). To lower this value we should increase q (the number of output samples calculated simultaneously) and decrease p. For minimum $p = 4$, I've measured performance for different q from 4 to 56. Results are shown in the table below:

q	registers used	memory reads per output sample	performance in Ms/s for N=1024
4	6	513.0	0.95
8	9	257.0	1.56
12	11	171.7	1.93
16	13	129.0	2.11
24	17	86.3	2.30
32	21	65.0	2.27
40	25	52.2	2.35
44	27	47.5	2.57
48	29	43.7	2.37
52	31	40.4	2.60
56	all	37.6	0.45

The number of used registers have been found by analysing executable code and checking a number of "1"-s OR-ed into VRSAVE special register used in multitasking environment for task switching. For $q = 56$ there is not enough AltiVec registers. The GCC 2.95.4 compiler used just emulates missing registers in memory, and as it can be seen in the disassembled code, generates a lot of load-store instructions, which of course degrades performance. Experiment results confirm expectations, the best results are achieved when number of simultaneously calculated output samples is as big as possible using all available SIMD registers, and for AltiVec architecture optimal block size is 4×52. The following graph compares performance of reference code, optimized scalar code (using available 32 FPU registers and 1×8 blocks) and optimized AltiVec code using 4×52 blocks. The source code for benchmark program is available in [13].



4. CONCLUSION

SIMD units are very common in nowadays personal computers, as almost every produced CPU has one built-in. Their computing power however is not often put to a good use. The main reason for this is that classic methods of optimization don't take memory bandwidth limits into account in spite it is a well known bottleneck for current CPUs and their SIMD units particularly. In this paper I've shown, that proper calculation reordering, which increases data locality, can speed even very simple "clasically-unoptimizable" algorithm (like convolution is) up, even if number of additions and multiplications is not changed. Optimized AltiVec FIR code is 8 to 14 times faster than scalar reference code, in spite of only 4-way AltiVec parallelism. As many DSP techniques are based on digital convolution, these can be accelerated as well.

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] CROCHIERE R. E., RABINER L. R., *Multirate Digital Signal Processing*, 76–91.

[7] [—], *32-bit Floating Point Real and Complex 16-tap FIR Filter Implemented Using Streaming SIMD Extensions*, Intel 1999.

[8] [—], *MPC7450 RISC Microprocessor Family Reference Manual*, Freescale Semiconductor 2005.

[9] [—], *AltiVec Real FIR [application note]*, Motorola 1998.

[10] [—], *AltiVec Technology Programming Interface Manual,* Motorola 1999.

[11] LYONS R. G., *Wprowadzenie do cyfrowego przetwarzania sygnałów [Understanding Digital Signal Processing]*, 1997, 398–399.

[12] 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.

[13] KRASZEWSKI G., *Source code for AltiVec optimized FIR filter and benchmark program*, http://teleinfo.pb.edu.pl/~krashan/altivec/fir/.