Grzegorz Kraszewski
<*krashan@teleinfo.pb.edu.pl*>
Białystok Technical University
Department of Electric Engineering

Performance Analysis of Alternative Structures for 16-bit Integer FIR Filter Implemented on AltiVec SIMD Processing Unit

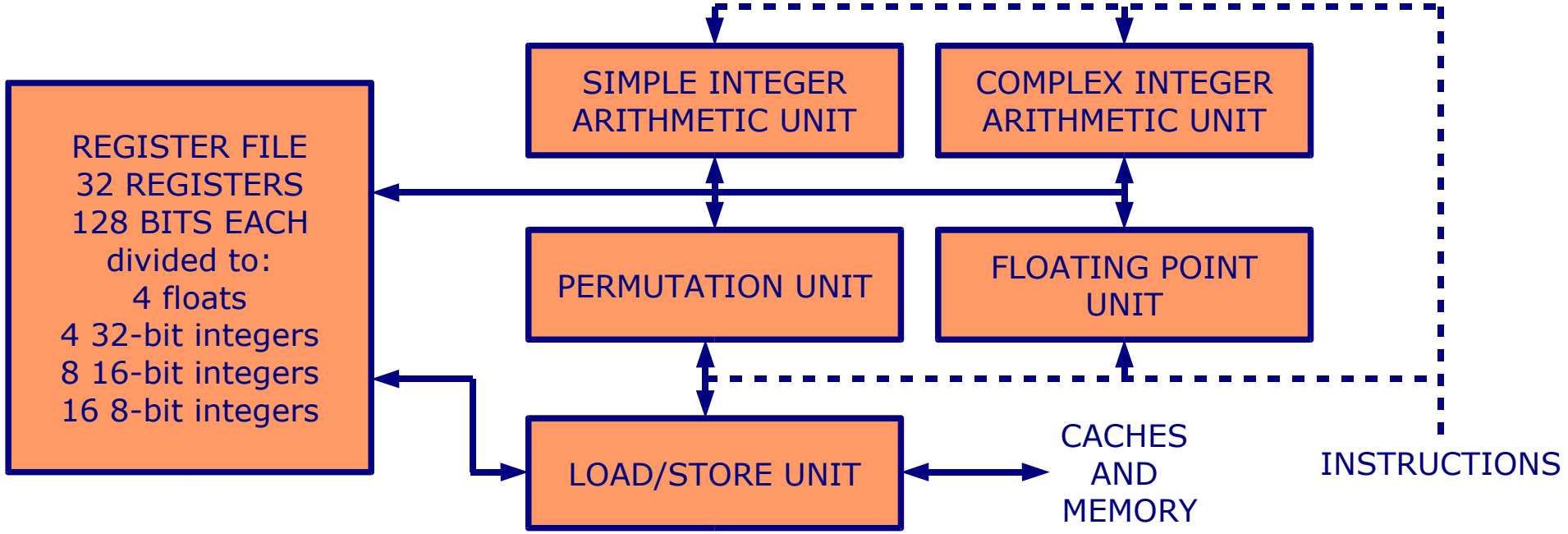
AltiVec

Presentation topics

1. A few words about AltiVec.
2. FIR filtering with a SIMD unit.
3. The *vec_msum()*, which does all the work.
4. Minimizing the number of memory accesses.
5. The first approach – data permutation at output.
6. The second one – data permutation at input.
7. The third approach – a golden compromise?
8. Experimental verification.
9. Analysis of execution units load.
10. Conclusion.

AltiVec processing unit

AltiVec is a name of SIMD unit implemented in microprocessors from **PowerPC** family, produced by Freescale (former Motorola SPS), IBM, AMCC and PA Semi. It may be compared to SSE on x86 architecture (in fact AltiVec has been introduced earlier). The latest AltiVec based product is **Cell** by IBM, its Synergistic Processing Elements are extension to the AltiVec design.



AltiVec

FIR filter and AltiVec

Assumptions: The filter is long (N > 100), input vector is very long (n > 10^6, possibly unlimited), these are typical assumptions for audio processing.

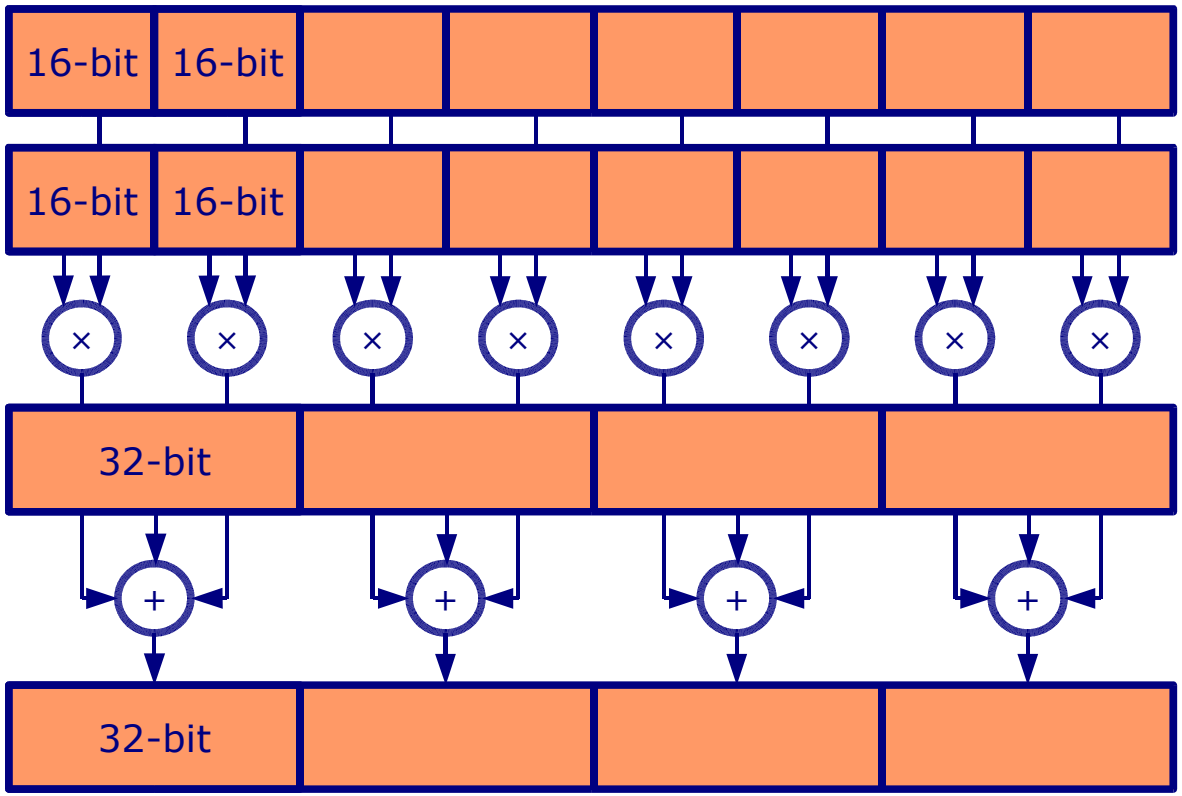
Starting from a basic formula $$y[k]=\sum_{n=0}^{N-1} x[k-n]\cdot f[n]$$

we can apply vectorization by simply multiplying and accumulating eight signal samples with eight filter coefficients in one *vec_msum()* instruction. On 1.0 GHz machine the theroetical computation speed limit is 8.0 Gtaps/s. Coming from theory into the practice we get only 0.9 Gtaps/s *(Pegasos II machine, PowerPC G4 7447)*.

The main problem is FIR filtering is memory bound task. The CPU will need 32 GB of data per second, main memory read speed (on test hardware) is only 225 MB/s. CPU caches improve the performance but not enough. Another problem is not-straight operand layout for *vec_msum()* instruction, which requires additional data permutation.

AltiVec

The vec_msum() instruction

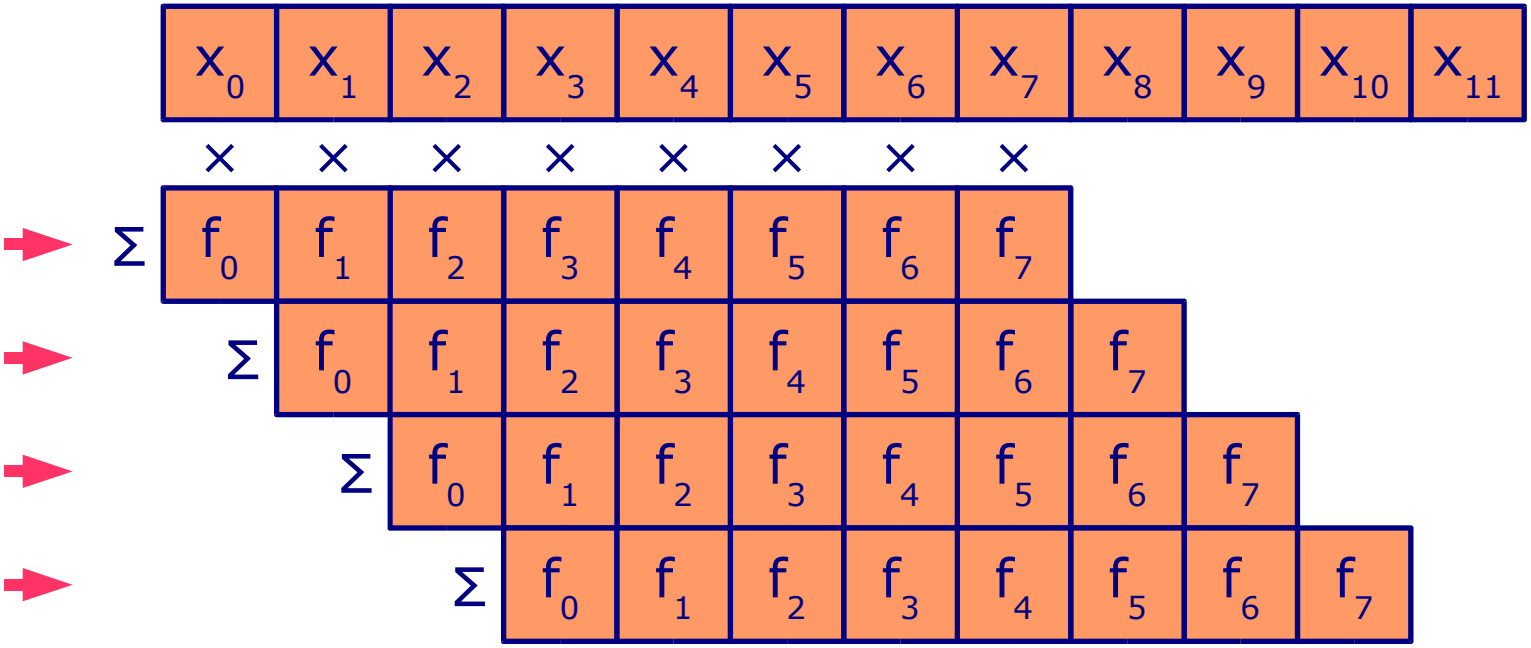


The instruction is designed to multiply and accumulate 16-bit operands with 32-bit accumulator. For FIR filter either input or filter vector (or both) are not in the straight order. This unusual layout is caused by different sizes of operands and the result.

Because of operation linearity, data permutation can be performed on input, on output or can be partitioned between input and output.

Minimizing memory accesses

AltiVec



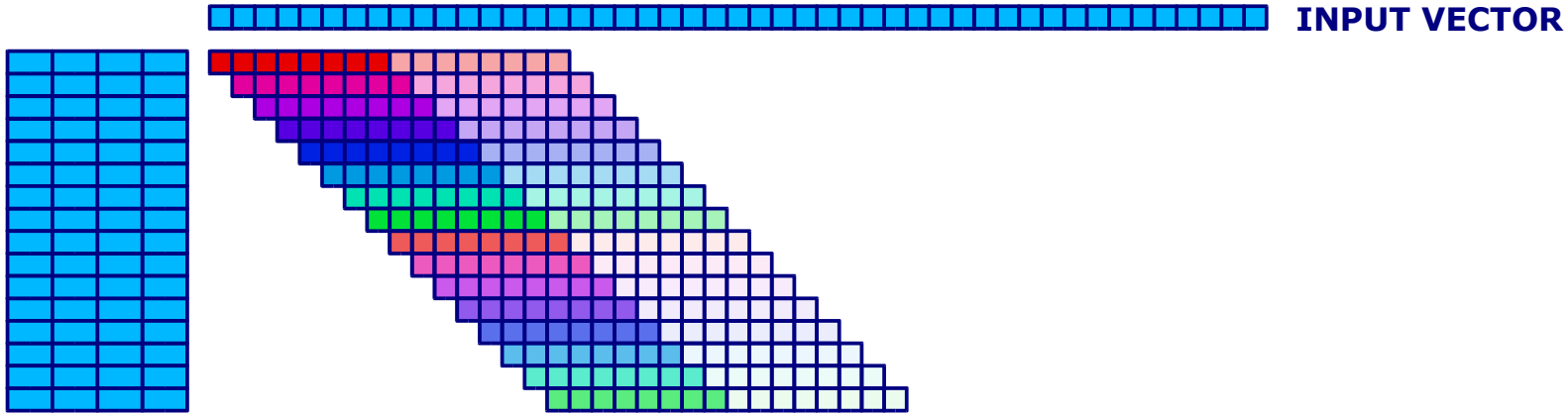
FIR filter calculation is no more than dot product of filter vector and shifted fragments of input vector. Doing it one by one output sample requires 2N memory reads (in terms of samples). The key to minimize memory accesses is to calculate many output samples at once (➔).

Calculating many output samples at once reduces memory reads by increasing data locality (every loaded number is reused as many times as possible). The upper limit for this parallelization is the number of available AltiVec registers. Let *N* be a number of filter taps, *m* be a number of output samples calculated in one go. Then one output sample requires number of memory reads given by:

$$R_{PS} = \frac{2N + m - 1}{m}$$

Data reordering at output

AltiVec

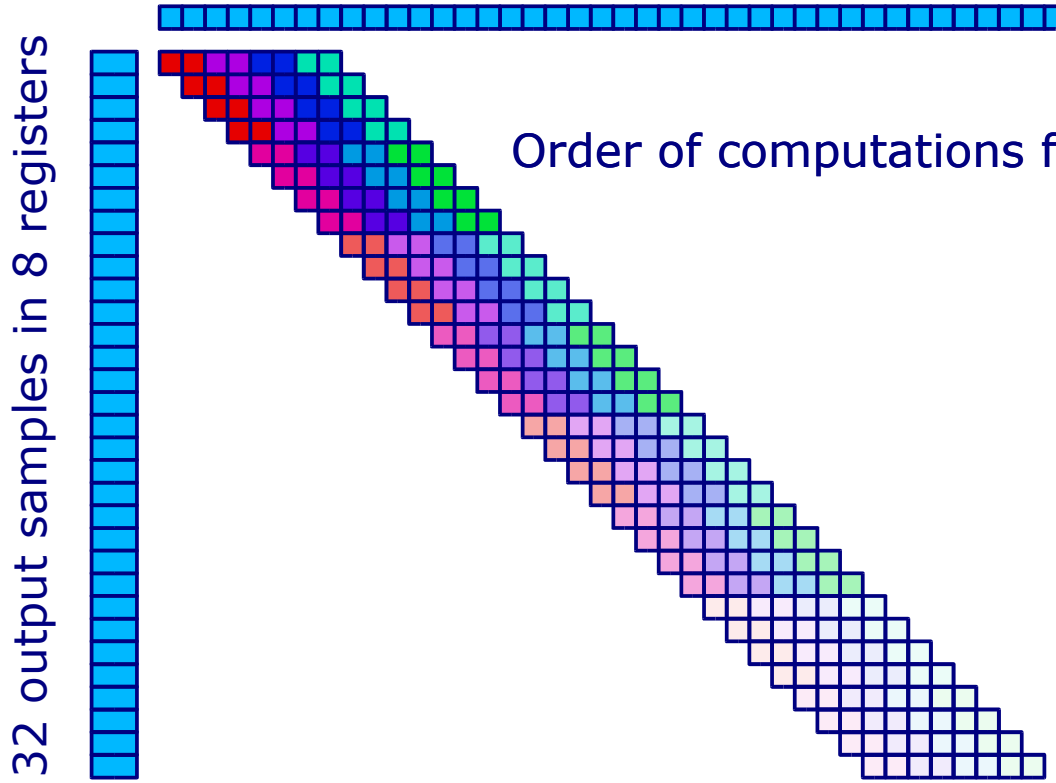


Order of computations from red to green.

In this implementation 8 consecutive input samples are multiplied with 8 consecutive filter coefficients. Every 8 filter coefficients are reused 16 times. At the start of the loop 24 samples are loaded, then shifted 8 times, after every 8 shifts the next 8 input samples are loaded. At the end of 16-sample horizontal pass every output sample is contained in one AltiVec register as 4 partial 32-bit sums. They have to be summed across the register, then rounded to 16-bit. Partial sums add 2 bits to the dynamic range of filter coefficients.

Data reordering at input

AltiVec

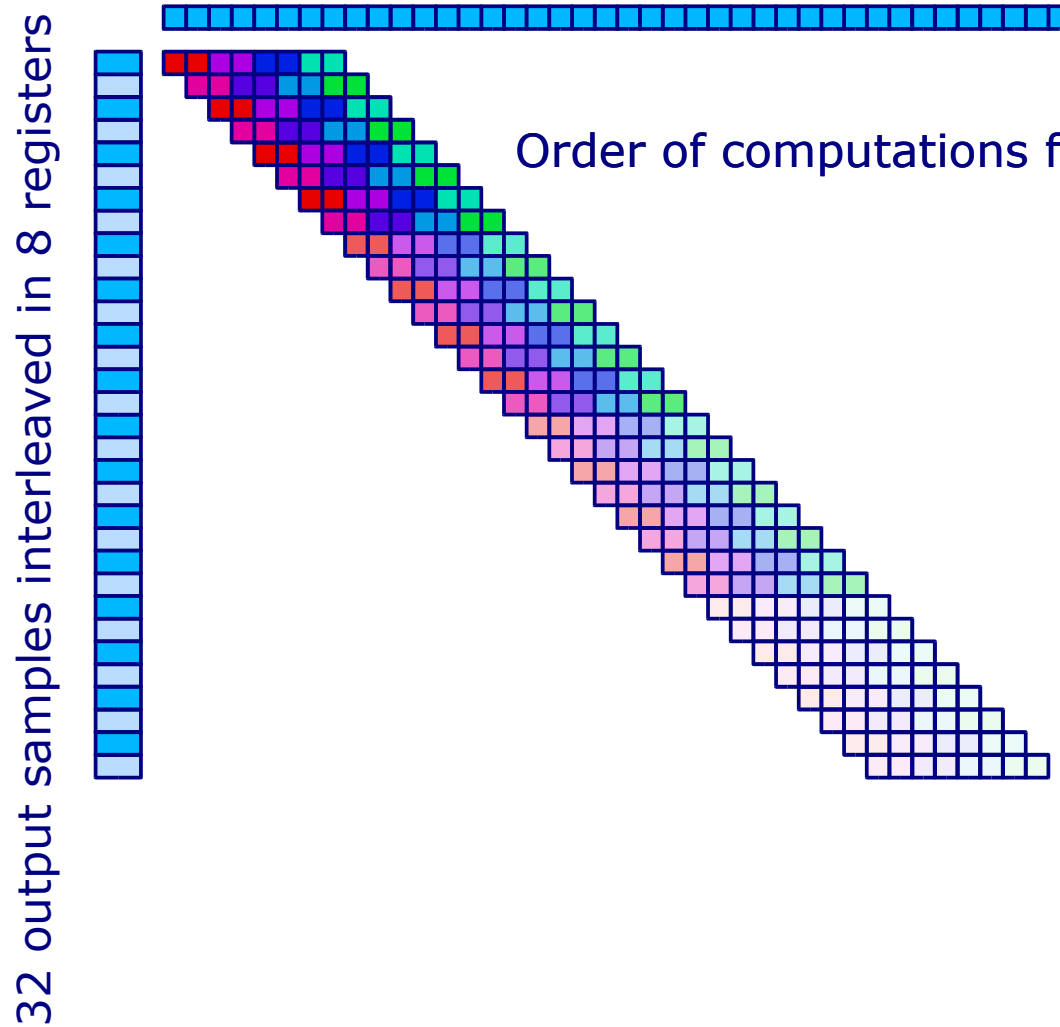


Order of computations from red to green.

In this structure all permutations are done on input data. The disadvantage is data have to be duplicated in registers (for example the first data vector has to be $[x_0 \; x_1 \; x_1 \; x_2 \; x_2 \; x_3 \; x_3 \; x_4]$, which imply more permutations per output sample. On the other hand there is no across-register summation of output, 4 outputs fit in one AltiVec register, which allows for 32-pipe processing.

The compromise

AltiVec

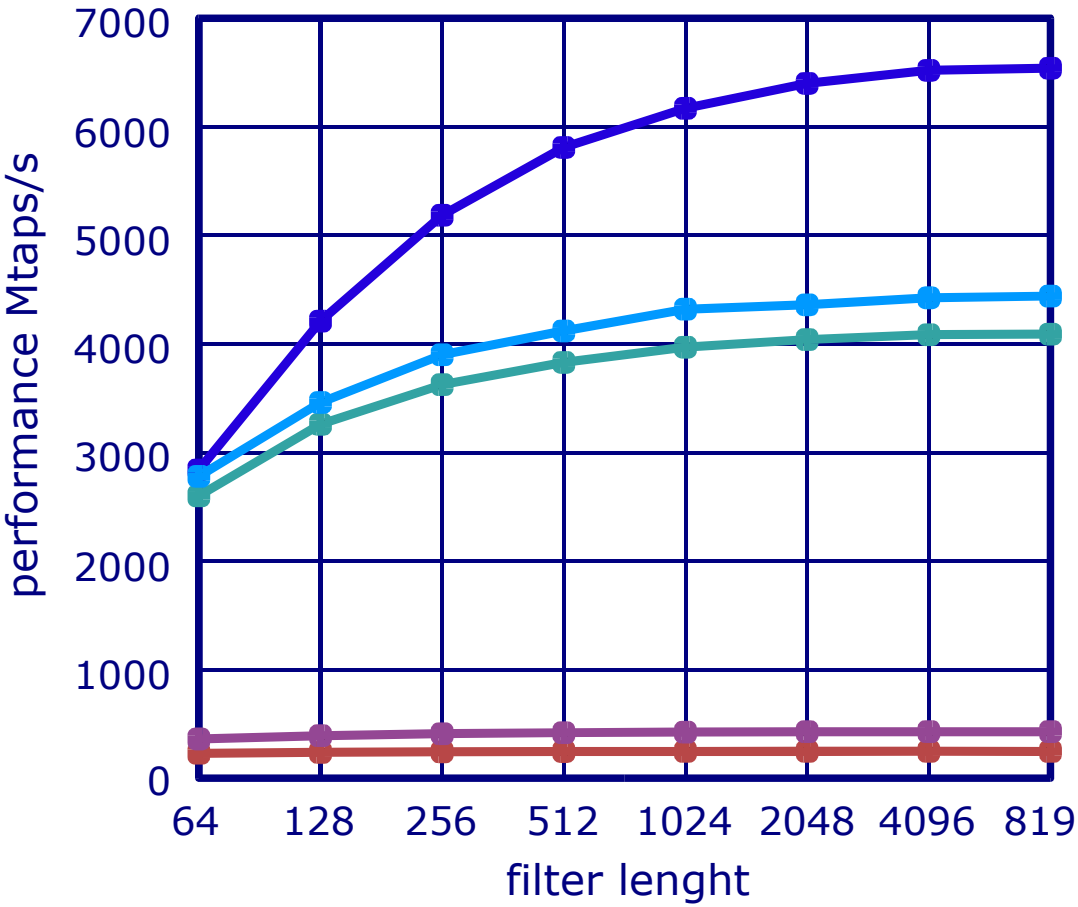


Order of computations from red to green.

This is a modification of the previous algorithm, even and odd samples of output are interleaved, so no input samples duplication is needed, at a cost of additional de-interleaving of output. De-interleaving may be performed in one operation with rounding output to 16 bits. This architecture preserves 32-pipe parallelism, so may be considered as the most promising in terms of performance.

Results of experiments

AltiVec



AltiVec A, 16-pipe, permutation on output.

AltiVec C, 32-pipe, permutation on both input and output.

AltiVec B, 32-pipe, permutation on input.

Scalar 8-pipe.

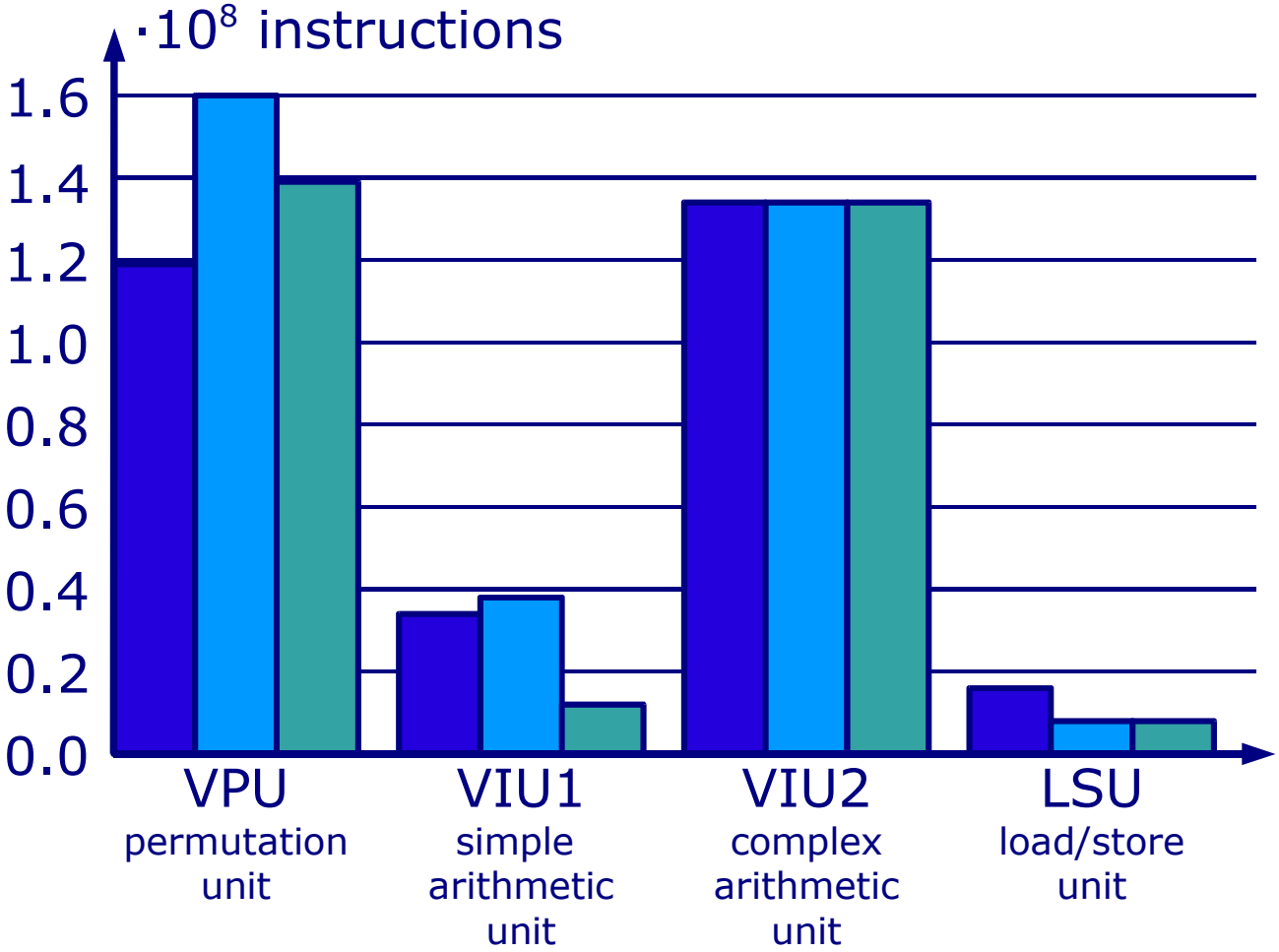
Scalar naive 1-pipe.

source code available at
http://teleinfo.pb.edu.pl/~krashan/altivec/fir16
results for Pegasos II, PowerPC G4/7447 1.0 GHz.

The best filter performed at **6.54 Gtaps/s**, which is 80% of theoretic throughput. It is unexpected that filter A, calculating only 16 samples per pass is the fastest. The explanation is optimization turned the task from memory bound into computation bound one.

AltiVec units load analysis

AltiVec



Number of instructions performed by AltiVec execution units while calculating 1024-tap filter on 1 048 576 (2^{20}) samples.

AltiVec A, 16-pipe, permutation on output.

AltiVec C, 32-pipe, permutation on both input and output.

AltiVec B, 32-pipe, permutation on input.

The load on permutation unit is what determines the performance of the three filter structures. While the structure "A" has only 16 pipes (so two times more memory loads, see the diagram), it puts less load on VPU. It is clear that now FIR filtering is a computationally bound task.

AltiVec

Conclusion

- Personal computers designs have a wide gap between CPU speed (especially its SIMD unit) and memory throughput.
- Classic approach to algorithms optimization (both hand-made and automatic by a compiler) does not take memory accesses into account.
- Increasing temporal data locality allows for significant algorithm speed-up without changing number of arithmetic operations. Exchanging some memory accesses for additional arithmetic can give even some more acceleration.
- Memory optimized algorithm for FIR filtering achieves 80% of theoretical CPU computing speed, which is not bad considering pipeline flushes at loops turns and the fact that data shifts (permutations) are not taken into account.