GCC2 versus GCC4 compiling AltiVec code

Grzegorz Kraszewski <krashan@teleinfo.pb.edu.pl>

1. Introduction

An official compiler for MorphOS operating system is still GCC 2.95.3. It is considered outdated by many people, and lack of newer GCC 3 or GCC 4 compilers is a reason for complaints. As some unofficial ports of GCC 3 and 4 appeared, there is an opportunity to test them and compare generated code. My main point of interest is AltiVec, so I've grabbed a port of GCC 4.0.3 done by Marcin "Morgoth" Kurek, and have given it a try with a *Reggae* class, *fir.filter* namely. For those of you not familiar with digital signal processing, FIR filtering is nothing more than doing a lot of MAC (multiply and accumulate) operations in a loop, so AltiVec is just what is needed to do it really fast. I've published a theory behind SIMD-optimized FIRs in [1] and [2]. I've just compiled the class with GCC 4, and ran some tests.

You may imagine how much I've been surprised when it turned out that **GCC 4.0.3 generated code is 5 to 15% slower** compared to GCC 2.95.3. I've extracted the important code from the class and written a testcase – still the same result. What is going on? The full source code of my benchmark is available in [3], the important part of the source is repeated here. I've compiled it as follows:

```
gcc4 -O2 -noixemul -maltivec -o intfir4 intfir.c

gcc -O2 -noixemul -fvec -c -o intfir2.o intfir.c
gcc -O2 -noixemul -fvec -o intfir2 intfir2.o saverest.o
```

A note for GCC 2 compilation – GCC 2.95.3 does not generate AltiVec non-scratch registers save and restore in a function prolog and epilog, it only generates calls to external functions. They are provided in *saverest.s* PowerPC assembler file, just copied from [4]. It should be noted however, these operations are done outside loops and have no impact on efficiency (the only difference is GCC 4.0.3 generates them inline automatically). Both versions are compiled from the same source.

Results of 16-bit integer FIR benchmark compiled with GCC 2.95.3 with AltiVec patches.

```
System:Stryszek/Devel/Work/mbench> intfir2
Table at $21B2CD10
Generated 100%.
Time elapsed:  0.205045 s [43.89 Msamples/s, 2809.14 Mtaps/s], 64 taps
Time elapsed:  0.277914 s [32.38 Msamples/s, 4145.17 Mtaps/s], 128 taps
Time elapsed:  0.448402 s [20.07 Msamples/s, 5138.25 Mtaps/s], 256 taps
Time elapsed:  0.796193 s [11.30 Msamples/s, 5787.54 Mtaps/s], 512 taps
Time elapsed:  1.499267 s [ 6.00 Msamples/s, 6147.00 Mtaps/s], 1024 taps
Time elapsed:  2.903064 s [ 3.10 Msamples/s, 6349.15 Mtaps/s], 2048 taps
Time elapsed:  5.674696 s [ 1.59 Msamples/s, 6496.21 Mtaps/s], 4096 taps
Time elapsed: 11.349737 s [ 0.79 Msamples/s, 6496.01 Mtaps/s], 8192 taps
```

Results of 16-bit integer FIR benchmark compiled with GCC 4.0.3. It is now 5 to 15 percent slower (!). The same code, different results. One may expect GCC 4 at least does not make it worse (if it can't make it better...), but it is not the case here.

```
System:Stryszek/Devel/Work/mbench> intfir4
Table at $21B8AF30
```

```
Generated 100%.
Time elapsed:  0.215843 s [41.70 Msamples/s, 2668.61 Mtaps/s], 64 taps
Time elapsed:  0.310773 s [28.96 Msamples/s, 3706.89 Mtaps/s], 128 taps
Time elapsed:  0.518138 s [17.37 Msamples/s, 4446.69 Mtaps/s], 256 taps
Time elapsed:  0.941178 s [ 9.56 Msamples/s, 4895.99 Mtaps/s], 512 taps
Time elapsed:  1.790672 s [ 5.03 Msamples/s, 5146.67 Mtaps/s], 1024 taps
Time elapsed:  3.471807 s [ 2.59 Msamples/s, 5309.05 Mtaps/s], 2048 taps
Time elapsed:  6.839173 s [ 1.32 Msamples/s, 5390.13 Mtaps/s], 4096 taps
Time elapsed: 13.642641 s [ 0.66 Msamples/s, 5404.23 Mtaps/s], 8192 taps
```

Something is definitely wrong. I've decided to disassemble the FIR routine and look into details (for the complete source code see [3]). Let's start with source:

2. The source code

```
void convolve_vector_mono_arch1_16pipe_int16(vector short *filter, vector short
  *source, vector short *dest, unsigned int frames, unsigned int taps)
{
  vector signed short x0, x1, x2, filter_block, t0, t1;
  vector signed int u0, u1, u2, u3, u4, u5, u6, u7;
  vector signed int u8, u9, uA, uB, uC, uD, uE, uF, zero, v0, v1;
  vector unsigned char p = (vector unsigned char) VEC_VALUE(0x02, 0x03, 0x04,
  0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x1C, 0x1D);
  unsigned int tapcounter;
  vector signed short *sp = NULL;
  vector short *fp;

  zero = vec_splat_s32(0);

  while (frames >= 16)
  {
    u0 = vec_splat_s32(0);    u1 = vec_splat_s32(0);
    u2 = vec_splat_s32(0);    u3 = vec_splat_s32(0);
    u4 = vec_splat_s32(0);    u5 = vec_splat_s32(0);
    u6 = vec_splat_s32(0);    u7 = vec_splat_s32(0);
    u8 = vec_splat_s32(0);    u9 = vec_splat_s32(0);
    uA = vec_splat_s32(0);    uB = vec_splat_s32(0);
    uC = vec_splat_s32(0);    uD = vec_splat_s32(0);
    uE = vec_splat_s32(0);    uF = vec_splat_s32(0);
    tapcounter = taps;
    sp = source;
    x0 = *sp++;    x1 = *sp++;
    fp = filter;

    while (tapcounter > 0)
    {
      filter_block = *fp++;
      x2 = *sp++;

      u0 = vec_msum(x0, filter_block, u0);
      t0 = vec_sld(x0, x1, 2);
      t1 = vec_sld(x0, x1, 4);
      u1 = vec_msum(t0, filter_block, u1);
      u2 = vec_msum(t1, filter_block, u2);
      t0 = vec_sld(x0, x1, 6);
      t1 = vec_sld(x0, x1, 8);
      u3 = vec_msum(t0, filter_block, u3);
      u4 = vec_msum(t1, filter_block, u4);
```

```
  t0 = vec_sld(x0, x1, 10);
  t1 = vec_sld(x0, x1, 12);
  u5 = vec_msum(t0, filter_block, u5);
  u6 = vec_msum(t1, filter_block, u6);
  t0 = vec_sld(x0, x1, 14);
  u7 = vec_msum(t0, filter_block, u7);

  u8 = vec_msum(x1, filter_block, u8);
  t0 = vec_sld(x1, x2, 2);
  t1 = vec_sld(x1, x2, 4);
  u9 = vec_msum(t0, filter_block, u9);
  uA = vec_msum(t1, filter_block, uA);
  t0 = vec_sld(x1, x2, 6);
  t1 = vec_sld(x1, x2, 8);
  uB = vec_msum(t0, filter_block, uB);
  uC = vec_msum(t1, filter_block, uC);
  t0 = vec_sld(x1, x2, 10);
  t1 = vec_sld(x1, x2, 12);
  uD = vec_msum(t0, filter_block, uD);
  uE = vec_msum(t1, filter_block, uE);
  t0 = vec_sld(x1, x2, 14);
  uF = vec_msum(t0, filter_block, uF);

  x0 = x1;
  x1 = x2;
  tapcounter -= 8;
}
x0 = vec_splat_s16(0);      x1 = vec_splat_s16(0);

v0 = vec_sums(u0, zero);    v1 = vec_sums(u1, zero);
x0 = vec_perm(x0, (vector signed short)v0, p);
x0 = vec_perm(x0, (vector signed short)v1, p);

v0 = vec_sums(u2, zero);    v1 = vec_sums(u3, zero);
x0 = vec_perm(x0, (vector signed short)v0, p);
x0 = vec_perm(x0, (vector signed short)v1, p);

v0 = vec_sums(u4, zero);    v1 = vec_sums(u5, zero);
x0 = vec_perm(x0, (vector signed short)v0, p);
x0 = vec_perm(x0, (vector signed short)v1, p);

v0 = vec_sums(u6, zero);    v1 = vec_sums(u7, zero);
x0 = vec_perm(x0, (vector signed short)v0, p);
x0 = vec_perm(x0, (vector signed short)v1, p);

v0 = vec_sums(u8, zero);    v1 = vec_sums(u9, zero);
x1 = vec_perm(x1, (vector signed short)v0, p);
x1 = vec_perm(x1, (vector signed short)v1, p);

v0 = vec_sums(uA, zero);    v1 = vec_sums(uB, zero);
x1 = vec_perm(x1, (vector signed short)v0, p);
x1 = vec_perm(x1, (vector signed short)v1, p);

v0 = vec_sums(uC, zero);    v1 = vec_sums(uD, zero);
x1 = vec_perm(x1, (vector signed short)v0, p);
x1 = vec_perm(x1, (vector signed short)v1, p);

v0 = vec_sums(uE, zero);    v1 = vec_sums(uF, zero);
x1 = vec_perm(x1, (vector signed short)v0, p);
x1 = vec_perm(x1, (vector signed short)v1, p);
```

```
    frames -= 16;
    *dest++ = x0;
    *dest++ = x1;
    source += 2
  }
}
```

The most important block consists of *vec_msum()* and *vec_sld()* instructions, as it is inside two nested loops.

3. GCC 2.95.3 executable

Here is a dissasembled integer FIR routine compiled with GCC2:

```
00001258 <convolve_vector_mono_arch1_16pipe_int16>:
    1258:   stwu     r1,-112(r1)
    125c:   mflr     r0
    1260:   stw      r0,116(r1)
    1264:   addi     r0,r1,96
    1268:   bl       3068 <_savev27>
    126c:   stw      r11,108(r1)
    1270:   oris     r11,r11,65535
    1274:   ori      r11,r11,61471
    1278:   mtspr    256,r11
```

This is a typical function prolog. Note that saving non-scratch AltiVec registers is not inlined, GCC 2 needs *_savevXX()* functions to be linked from separate object (compiled from assembler source taken from *AltiVec PIM* document). Looking at what is written in VRSAVE, we see, there are 25 AltiVec registers used.

```
    127c:   lis         r9,0
    1280:   vspltisw    v9,0
    1284:   cmplwi      r6,15
    1288:   addi        r9,r9,0
    128c:   vsldoi      v27,v9,v9,0
    1290:   lvx         v8,r0,r9
    1294:   ble         1434 <convolve_vector_mono_arch1_16pipe_int16+0x1dc>
    1298:   vsldoi      v5,v27,v27,0
    129c:   addi        r9,r4,16
    12a0:   mr.         r0,r7
    12a4:   lvx         v10,r0,r4
    12a8:   lvx         v11,r0,r9
    12ac:   addi        r6,r6,-16
    12b0:   addi        r4,r4,32
    12b4:   vsldoi      v6,v5,v5,0
    12b8:   addi        r9,r9,16
    12bc:   mr          r11,r3
    12c0:   vsldoi      v18,v5,v5,0
    12c4:   addi        r10,r5,16
    12c8:   vsldoi      v7,v6,v6,0
    12cc:   vor         v17,v6,v6
    12d0:   vsldoi      v4,v7,v7,0
    12d4:   vor         v16,v7,v7
    12d8:   vsldoi      v3,v4,v4,0
    12dc:   vor         v15,v4,v4
    12e0:   vsldoi      v2,v3,v3,0
```

```
12e4:   vor         v14,v3,v3
12e8:   vsldoi      v19,v2,v2,0
12ec:   vor         v30,v2,v2
12f0:   vsldoi      v31,v19,v19,0
12f4:   vor         v29,v19,v19
12f8:   vsldoi      v28,v31,v31,0
```

The main thing here is (except loop organization) zeroing 16 AltiVec registers used as accumulators. The source have just *vec_splat_s32()* repeated 16 times, but GCC 2 cleverly does just one *vspltisw* at $1280 and then copies *v9* to other 15 registers using *vor* and *vsldoi* alternately to balance load between VPU (permutation unit), executing *vsldoi* and VIU1 (simple integer arithmetic unit), executing *vor*.

```
12fc:   beq         1398 <convolve_vector_mono_arch1_16pipe_int16+0x140>
1300:   lvx         v0,r0,r11
1304:   vsldoi      v13,v10,v11,2
1308:   addic.      r0,r0,-8
130c:   vsldoi      v12,v10,v11,4
1310:   lvx         v1,r0,r9
1314:   addi        r11,r11,16
1318:   addi        r9,r9,16
131c:   vmsumshm    v5,v13,v0,v5
1320:   vsldoi      v13,v10,v11,6
1324:   vmsumshm    v17,v12,v0,v17
1328:   vsldoi      v12,v10,v11,8
132c:   vmsumshm    v18,v10,v0,v18
1330:   vmsumshm    v6,v13,v0,v6
1334:   vsldoi      v13,v10,v11,10
1338:   vmsumshm    v16,v12,v0,v16
133c:   vsldoi      v12,v10,v11,12
1340:   vmsumshm    v14,v11,v0,v14
1344:   vmsumshm    v7,v13,v0,v7
1348:   vsldoi      v13,v10,v11,14
134c:   vmsumshm    v15,v12,v0,v15
1350:   vsldoi      v12,v11,v1,4
1354:   vor         v10,v11,v11
1358:   vmsumshm    v4,v13,v0,v4
135c:   vsldoi      v13,v11,v1,2
1360:   vmsumshm    v30,v12,v0,v30
1364:   vsldoi      v12,v11,v1,8
1368:   vmsumshm    v3,v13,v0,v3
136c:   vsldoi      v13,v11,v1,6
1370:   vmsumshm    v29,v12,v0,v29
1374:   vsldoi      v12,v11,v1,12
1378:   vmsumshm    v2,v13,v0,v2
137c:   vsldoi      v13,v11,v1,10
1380:   vmsumshm    v28,v12,v0,v28
1384:   vmsumshm    v19,v13,v0,v19
1388:   vsldoi      v13,v11,v1,14
138c:   vor         v11,v1,v1
1390:   vmsumshm    v31,v13,v0,v31
1394:   bne         1300 <convolve_vector_mono_arch1_16pipe_int16+0xa8>
```

The sequence above is the critical part of code, as it is inside both the internal and external loop. From results of tests it is clear, that GCC 2 compiled code is significantly faster than GCC 4 one. Why? I'll show it later, when analysing temporary variables *t0* and *t1* usage pattern (*v12* and *v13* here).

```
1398:   vsumsws     v1,v18,v9
139c:   cmplwi      r6,15
```

```
13a0:   vsumsws     v0,v5,v9
13a4:   vsldoi      v11,v27,v27,0
13a8:   vperm       v10,v11,v1,v8
13ac:   vsumsws     v1,v17,v9
13b0:   vperm       v10,v10,v0,v8
13b4:   vsumsws     v0,v6,v9
13b8:   vperm       v10,v10,v1,v8
13bc:   vsumsws     v1,v16,v9
13c0:   vperm       v10,v10,v0,v8
13c4:   vsumsws     v0,v7,v9
13c8:   vperm       v10,v10,v1,v8
13cc:   vsumsws     v1,v15,v9
13d0:   vperm       v10,v10,v0,v8
13d4:   vsumsws     v0,v4,v9
13d8:   vperm       v10,v10,v1,v8
13dc:   vsumsws     v1,v14,v9
13e0:   vperm       v10,v10,v0,v8
13e4:   vsumsws     v0,v3,v9
13e8:   stvx        v10,r0,r5
13ec:   mr          r5,r10
13f0:   vperm       v11,v11,v1,v8
13f4:   vsumsws     v1,v30,v9
13f8:   vperm       v11,v11,v0,v8
13fc:   vsumsws     v0,v2,v9
1400:   vperm       v11,v11,v1,v8
1404:   vsumsws     v1,v29,v9
1408:   vperm       v11,v11,v0,v8
140c:   vsumsws     v0,v19,v9
1410:   vperm       v11,v11,v1,v8
1414:   vsumsws     v1,v28,v9
1418:   vperm       v11,v11,v0,v8
141c:   vsumsws     v0,v31,v9
1420:   vperm       v11,v11,v1,v8
1424:   vperm       v11,v11,v0,v8
1428:   stvx        v11,r0,r5
142c:   addi        r5,r5,16
1430:   bgt         1298 <convolve_vector_mono_arch1_16pipe_int16+0x40>
```

This part is responsible for summing partial results across accumulators (*vsumsws*, 16 times), extracting most significant 16 bits from accumulators, and then interleaving data before storing (*vperm*). As this code is outside the inner loop its performance is less critical.

```
1434:   lwz         r10,108(r1)
1438:   addi        r0,r1,96
143c:   bl          30cc <_restv27>
1440:   lwz         r0,116(r1)
1444:   mtlr        r0
1448:   addi        r1,r1,112
144c:   blr
```

The function epilog. Restore registers, VRSAVE and stack, then *blr* to the caller.

4. GCC 4.0.3 executable

Ok, now let's look at the same code compiled with GCC 4.0.3:

```
00001320 <convolve_vector_mono_arch1_16pipe_int16>:
    1320:   stwu        r1,-112(r1)
    1324:   li          r0,16
    1328:   stvx        v27,r1,r0
    132c:   li          r0,32
    1330:   stvx        v28,r1,r0
    1334:   li          r0,48
    1338:   stvx        v29,r1,r0
    133c:   li          r0,64
    1340:   stvx        v30,r1,r0
    1344:   li          r0,80
    1348:   stvx        v31,r1,r0
    134c:   mfspr       r0,256
    1350:   stw         r0,108(r1)
    1354:   oris        r0,r0,65535
    1358:   ori         r0,r0,61471
    135c:   mtspr       256,r0
```

This is again the function prolog. Nothing special, but saving non-scratch AltiVec registers is now inlined, as automatically generated by the compiler. Number of AltiVec registers used is exactly the same (25) as in GCC 2 version, we can also notice, that there are the same registers used.

```
    1360:   vspltisw    v18,0
    1364:   cmplwi      cr7,r6,15
    1368:   mflr        r0
    136c:   stw         r0,116(r1)
    1370:   bgt         cr7,13b4 <convolve_vector_mono_arch1_16pipe_int16+0x94>
```

The first significant difference in the loop organization. GCC 2 generates jump at loop exit (just like in the source), GCC 4 prefers to jump at every loop turn, so then we should jump as well to the offset $13B4. The external loop is controlled by sample counter located at *r6*. The *v18* register is zeroed, it will be used later for clearing other 15 registers used as accumulators.

```
    1374:   li          r0,16
    1378:   lwz         r12,108(r1)
    137c:   lvx         v27,r1,r0
    1380:   li          r0,32
    1384:   lvx         v28,r1,r0
    1388:   li          r0,48
    138c:   lvx         v29,r1,r0
    1390:   li          r0,64
    1394:   lvx         v30,r1,r0
    1398:   li          r0,80
    139c:   lvx         v31,r1,r0
    13a0:   mtspr       256,r12
    13a4:   lwz         r0,116(r1)
    13a8:   addi        r1,r1,112
    13ac:   mtlr        r0
    13b0:   blr
```

The code fragment above is the function eplilog, non-scratch AltiVec registers are restored, as well as VRSAVE register and the stack. Final *blr* returns to the caller.

```
13b4:   lis         r9,0
13b8:   cmpwi       cr6,r7,0
13bc:   addi        r9,r9,0
13c0:   li          r8,16
13c4:   lvx         v17,r0,r9
13c8:   vspltish    v27,0
13cc:   b           146c <convolve_vector_mono_arch1_16pipe_int16+0x14c>
```

GCC 4 liked to insert an unconditional branch here (note that GCC 2 does not need it). Then we should jump in our code analyse to the offset $146C. A *lvx* at $13C4 loads permutation control vector for *vec_perm()* used later. An internal loop, controlled by filter tap counter (located at *r7*) is organized here.

```
13d0:   addi        r6,r6,-16
13d4:   vsumsws     v0,v4,v18
13d8:   cmplwi      cr7,r6,15
13dc:   vsumsws     v13,v28,v18
13e0:   vperm       v0,v27,v0,v17
13e4:   vsumsws     v12,v7,v18
13e8:   vperm       v0,v0,v13,v17
13ec:   vsumsws     v11,v30,v18
13f0:   vperm       v0,v0,v12,v17
13f4:   vsumsws     v10,v9,v18
13f8:   vperm       v0,v0,v11,v17
13fc:   vsumsws     v9,v31,v18
1400:   vperm       v0,v0,v10,v17
1404:   vsumsws     v8,v8,v18
1408:   vperm       v0,v0,v9,v17
140c:   vsumsws     v7,v14,v18
1410:   vperm       v0,v0,v8,v17
1414:   vsumsws     v1,v6,v18
1418:   vperm       v0,v0,v7,v17
141c:   vsumsws     v6,v15,v18
1420:   stvx        v0,r0,r5

1424:   vsumsws     v5,v5,v18
1428:   vperm       v1,v27,v1,v17
142c:   vsumsws     v4,v16,v18
1430:   vperm       v1,v1,v6,v17
1434:   vsumsws     v3,v3,v18
1438:   vperm       v1,v1,v5,v17
143c:   vsumsws     v2,v2,v18
1440:   vperm       v1,v1,v4,v17
1444:   vsumsws     v19,v19,v18
1448:   vperm       v1,v1,v3,v17
144c:   vsumsws     v0,v29,v18
1450:   vperm       v1,v1,v2,v17
1454:   mr          r4,r10
1458:   vperm       v1,v1,v19,v17
145c:   vperm       v1,v1,v0,v17
1460:   stvx        v1,r5,r8
```

Across-register accumulator summing, truncating 16 least significant bits, merging.

```
1464:   addi        r5,r5,32
1468:   ble-        cr7,1374 <convolve_vector_mono_arch1_16pipe_int16+0x54>
```

External (controlled by output sample counter) loop end.

```
146c:   addi        r10,r4,32
1470:   vor         28,v18,v18
1474:   vor         v4,v18,v18
1478:   vor         v30,v18,v18
147c:   vor         v7,v18,v18
1480:   vor         v31,v18,v18
1484:   vor         v9,v18,v18
1488:   vor         v14,v18,v18
148c:   vor         v8,v18,v18
1490:   vor         v15,v18,v18
1494:   vor         v6,v18,v18
1498:   vor         v16,v18,v18
149c:   vor         v5,v18,v18
14a0:   vor         v2,v18,v18
14a4:   vor         v3,v18,v18
14a8:   vor         v29,v18,v18
14ac:   vor         v19,v18,v18
```

Well, we are at the first GCC 4 problem. Although it was smart enough to replace a series of 15 *vec_splat()* with register copying, it does the copy only with one kind of instruction, possibly saturating VIU1 pipeline. This is not critical however, instructions do not depend on each other (so no pipeline stalls) and this sequence is done only once per external loop turn.

```
14b0:   lvx         v10,r0,r4
14b4:   lvx         v11,r4,r8
```

Loading the first 16 samples of input vector.

```
14b8:   beq   cr6,13d0<convolve_vector_mono_arch1_16pipe_int16+0xb0>
```

This time GCC 4 did not reorganized the loop. The above conditional branch is a part of internal loop, controlled by filter counter in *r7* (look at *cmpwi* at $13B8). What is funny, the jump is taken back, while in the source it is forward, as usual with loops.

```
14bc:   mr          r11,r10
14c0:   mr          r0,r7
14c4:   mr          r9,r3
14c8:   addic.      r0,r0,-8
14cc:   lvx         v13,r0,r9
14d0:   lvx         v12,r0,r11
```

The first fragment of the internal loop, two *lvx* load filter coefficents. Note that they are not interleaved with some other instructions as one may expect.

```
14d4:   vsldoi      v0,v10,v11,14
14d8:   vsldoi      v1,v10,v11,4
14dc:   vmsumshm    v14,v0,v13,v14
14e0:   vmsumshm    v4,v10,v13,v4
14e4:   vsldoi      v0,v10,v11,2
14e8:   vmsumshm    v6,v11,v13,v6
14ec:   vmsumshm    v28,v0,v13,v28
14f0:   vmsumshm    v7,v1,v13,v7
14f4:   vsldoi      v0,v10,v11,6
14f8:   vsldoi      v1,v10,v11,8
14fc:   vmsumshm    v30,v0,v13,v30
1500:   vmsumshm    v9,v1,v13,v9
1504:   vsldoi      v0,v10,v11,10
```

```
1508:   vsldoi      v1,v10,v11,12
150c:   vmsumshm    v31,v0,v13,v31
1510:   vmsumshm    v8,v1,v13,v8
1514:   vsldoi      v0,v11,v12,2
1518:   vsldoi      v1,v11,v12,4
151c:   vor         v10,v11,v11
1520:   vmsumshm    v15,v0,v13,v15
1524:   vmsumshm    v5,v1,v13,v5
1528:   vsldoi      v0,v11,v12,6
152c:   vsldoi      v1,v11,v12,8
1530:   vmsumshm    v16,v0,v13,v16
1534:   vmsumshm    v3,v1,v13,v3
1538:   vsldoi      v0,v11,v12,10
153c:   vsldoi      v1,v11,v12,12
1540:   vmsumshm    v2,v0,v13,v2
1544:   addi        r9,r9,16
1548:   vsldoi      v0,v11,v12,14
154c:   addi        r11,r11,16
1550:   vmsumshm    v19,v1,v13,v19
1554:   vor         v11,v12,v12
1558:   vmsumshm    v29,v0,v13,v29
155c:   bne         14c8 <convolve_vector_mono_arch1_16pipe_int16+0x1a8>
1560:   b           13d0 <convolve_vector_mono_arch1_16pipe_int16+0xb0>
```

5. Where is the problem?

The problem is caused by rescheduling of AltiVec instructions by GCC 4. Version 2 of the compiler puts AltiVec instructions just in the same order as they stand in the source. While it may be considered a disadvantage for unexperienced programmer writing expressions in random order, any hand-made odrering is ruined. The order of instructions, I've used in the code is not a rocket science, it is based just on a basic knowledge of how modern microprocessors work, what is pipeline, how instructions are distributed between execution units, etc. GCC 4 preferred to "know better", but the final result is wrong. Let's look at usage of temporary variables *t0* and *t1*. GCC 2 placed these variables in *v13* and *v12*, GCC 4 preferred *v0* and *v1*.

○ – register used as source
● – register used as destination

GCC 2.95.3					GCC 4.0.3				
Offset	**v12**	**v13**	**VIU2**	**VPU**	**Offset**	**v0**	**v1**	**VIU2**	**VPU**
$1300					$14D4	●			●
$1304		●		●	$14D8	○	●		●
$1308					$14DC		○	●	
$130C	●			●	$14E0			●	
$1310					$14E4	●			●
$1314					$14E8			●	
$1318					$14EC	○		●	
$131C		○	●		$14F0		○	●	
$1320		●		●	$14F4	●			●
$1324	○		●		$14F8		●		●
$1328	●			●	$14FC	○		●	
$132C			●		$1500		○	●	
$1330		○	●		$1504	●			●
$1334		●		●	$1508		●		●
$1338	○		●		$150C	○		●	

GCC 2.95.3					GCC 4.0.3				
Offset	v12	v13	VIU2	VPU	Offset	v0	v1	VIU2	VPU
$133C	●			●	$1510		○	●	
$1340		↓	●		$1514	●			●
$1344		○	●		$1518		●		●
$1348		●		●	$151C				
$134C	○		●		$1520	○	↓	●	
$1350	●			●	$1524		○	●	
$1354		↓			$1528	●			●
$1358		○	●		$152C		●		●
$135C		●		●	$1530	○	↓	●	
$1360	○		●		$1534		○	●	
$1364	●	↓		●	$1538	●			●
$1368		○	●		$153C	↓	●		●
$136C	↓	●		●	$1540	○	↓	●	
$1370	○		●		$1544				
$1374	●	↓		●	$1548	●	↓		●
$1378	↓	○	●		$154C	↓	↓		
$137C	↓	●		●	$1550	↓	○	●	
$1380	○	↓	●		$1554	↓			
$1384		○	●		$1558	○		●	
$1388		●		●	$155C				
$138C		↓							
$1390		○	●						

My hand-made scheduling was done with one thing in mind – AltiVec has pipelined execution units, both *vsldoi* and *vmsumshm* have 2 and 4 cycles latency respectively and 1 cycle throughput. It means instruction which produces a result, and the one using it should be separated to avoid pipeline stalls. In most cases there is at least one additional VIU2 instruction between *vsldoi* generating a result (which is executed by VPU) and *vmsumshm* using it (which is executed by VIU2). What is easily visible, a third temporary variable should improve the performance a bit, I will for sure test it in the future. GCC 4 managed to get the code a bit shorter (by moving some auxiliary instructions out of the critical block), but destination-source separation is definitely worse (red arrows in the table). GCC 4 seems to try to generate an uniform pattern here, but it hits performance at the end. I don't know why it uses such a strange scheduling, what I can say, it simply does not work. Maybe there are some compiler options able to improve the code quality, but generally I'm disappointed with GCC 4. One may say I can improve GCC4 code by tweaking the source code, or messing with compiler options. Well, possible, but this is not a point. When I migrate from an older version of a tool to a newer one, I expect this new wersion will give me at least as good results as the old one (possibly even better). GCC 4 breaks this rule, working worse.

6. References

[1] KRASZEWSKI G., *Performance Analysis of Alternative Structures for 16-bit Integer FIR Filter Implemented on AltiVec SIMD Processing Unit*, Proceedings of IEEE Workshop on Signal Processing, Poznań, 2006, 83–87.

[2] KRASZEWSKI G., *Fast FIR Filters for SIMD Processors With Limited Memory Bandwidth*, Proceedings of XI Symposium AES „New Trends in Audio and Video", Białystok, 2006, 467–472.

[3] KRASZEWSKI G., *Testcase for GCC 2.95.3 and GCC 4.0.3 Compilers Compiling AltiVec Code*, http://teleinfo.pb.edu.pl/~krashan/altivec/gccbenchmark/.

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