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 It is considered outdated by many people, and lack of newer GCC 3 or GCC 4 compilers is a reason for complaints.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 -c -o intfir2.o 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: 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

```c
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);
    vector short *fp;

    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 disassembled 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: vsplitisw 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
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
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 vsplitisw 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.

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).
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.

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:

```assembly
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

    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>
```

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.

```assembly
    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.

```assembly
    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 epilog, non-scratch AltiVec registers are restored, as well as VRSAVE register and the stack. Final blr returns to the caller.
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.

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

External (controlled by output sample counter) loop end.
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_monarch16pipe_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 coefficients. 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
```
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 ordering 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>v12</th>
<th>v13</th>
<th>VIU2</th>
<th>VPU</th>
<th>Offset</th>
<th>v0</th>
<th>v1</th>
<th>VIU2</th>
<th>VPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1300</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$14D4</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1304</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>$14D8</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1308</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$14DC</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$130C</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$14E0</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1310</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$14E4</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$14E8</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1318</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$14EC</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$131C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$14F0</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1320</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$14F4</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1324</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$14F8</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1328</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>$14FC</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$132C</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$1500</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1330</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>$1504</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1334</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>$1508</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1338</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>$150C</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 version will give me at least as good results as the old one (possibly even better). GCC 4 breaks this rule, working worse.

6. References


