Vectorization of Algorithm Adler32 Using Altivec

Konstantinos Margaritis, markos@debian.org

May 15, 2005

Nafplion, Greece

Summary

This paper describes the process of vectorization of the Adler32 hashing/checksum algorithm using Altivec. Altivec is the vector unit found in some PowerPC processors, namely the models commonly referred to as G4 and G5. The process of vectorization of an algorithm is not always possible, but when it is, it usually offers significant speed increases. The vectorized Adler32 algorithm was about 2-2.5 times faster than the scalar.

I'll refrain from presenting the actual algorithm in all its rigorous mathematics, I'm sure you can find it in some mathematical journal. Instead I'll present the only form of the algorithm I found worth working on, the one found in zlib. Zlib is a library comprised of common routines for compression/uncompression and some hashing algorithms. Its main purpose is to be used in cases where LZW compression/uncompression is needed, such as gzip/zip. The Adler32 algorithm is used by these routines for checksum checks.

The form in zlib is easily vectorizable at least on first site. Actually it is, but its vectorization with Altivec has some limitations as we will see soon. This is the form of the algorithm in `adler32.c`:

```
#define BASE 65521UL    /* largest prime smaller than 65536 */
#define NMAX 5552
/* NMAX is the largest n such that 255n(n+1)/2 + (n+1)(BASE-1) <= 2^32-1 */

#define DO1(buf,i)  {s1 += buf[i]; s2 += s1;}
#define DO2(buf,i)  DO1(buf,i); DO1(buf,i+1);
#define DO4(buf,i)  DO2(buf,i); DO2(buf,i+2);
#define DO8(buf,i)  DO4(buf,i); DO4(buf,i+4);
#define DO16(buf)   DO8(buf,0); DO8(buf,8);

#ifdef NO_DIVIDE
#  define MOD(a) \
    do { \
        if (a >= (BASE << 16)) a -= (BASE << 16); \
        if (a >= (BASE << 15)) a -= (BASE << 15); \
        if (a >= (BASE << 14)) a -= (BASE << 14); \
        if (a >= (BASE << 13)) a -= (BASE << 13); \
        if (a >= (BASE << 12)) a -= (BASE << 12); \
        if (a >= (BASE << 11)) a -= (BASE << 11); \
        if (a >= (BASE << 10)) a -= (BASE << 10); \
        if (a >= (BASE << 9)) a -= (BASE << 9); \
        if (a >= (BASE << 8)) a -= (BASE << 8); \
        if (a >= (BASE << 7)) a -= (BASE << 7); \
        if (a >= (BASE << 6)) a -= (BASE << 6); \
        if (a >= (BASE << 5)) a -= (BASE << 5); \
        if (a >= (BASE << 4)) a -= (BASE << 4); \
        if (a >= (BASE << 3)) a -= (BASE << 3); \
        if (a >= (BASE << 2)) a -= (BASE << 2); \
        if (a >= (BASE << 1)) a -= (BASE << 1); \
```

```
        if (a >= BASE) a -= BASE; \
    } while (0)
#else
#  define MOD(a) a %= BASE
#endif

uLong ZEXPORT adler32(adler, buf, len)
    uLong adler;
    const Bytef *buf;
    uInt len;
{
    unsigned long s1 = adler & 0xffff;
    unsigned long s2 = (adler >> 16) & 0xffff;
    int k;

    if (buf == Z_NULL) return 1L;

    while (len > 0) {
        k = len < NMAX ? (int)len : NMAX;
        len -= k;
        while (k >= 16) {
            DO16(buf);
            buf += 16;
            k -= 16;
        }
        if (k != 0) do {
            s1 += *buf++;
            s2 += s1;
        } while (--k);
        MOD(s1);
        MOD(s2);
    }
    return (s2 << 16) | s1;
}
```

We plainly see that the main loop worth vectorizing is this:

```
        while (k >= 16) {
            DO16(buf);
            buf += 16;
            k -= 16;
        }
```

where `DO16(buf)` is a macro as we can see. The algorithm basically needs to calculate the sums `s1` and `s2`, get their modulus to 65521 and then continue with the summations. Originally the algorithm had to calculate the modulo every step, but this revised form only does the modulo calculation, every 5552 iterations. Why this number? Since `s2` is an unsigned long int (thus with an upper limit of `2^32-1`) we have to stop the summation before we have an overflow. How can we calculate this? Assume that our buffer is filled with bytes with the value 0xFF (255), the maximum value for an unsigned char. Then `s2` will get filled up pretty quickly, and we can see that it will exceed the limit `2^32-1` after 5552 iterations, basically we have to solve the inequality:

$$255\frac{1}{2}N_{max}(N_{max}+1)+(N_{max}+1)(BASE-1)\leq 2^{32}-1$$

It is easily found that `NMAX` is 5552 for the upper limit of `2^32 -1`.

Now let's look more closely at how the `s1` and `s2` integers (out of which in the end the Adler32 checksum is constructed) are calculated. Let's take the calculations in steps (we have assumed that `c[i]` is the character of the buffer in the position i):

$$s_{1_1} = s_{1_0} + c[0]$$
$$s_{2_1} = s_{2_0} + s_{1_1} = s_{2_0} + s_{1_0} + c[0]$$
$$s_{1_2} = s_{1_1} + c[1] = s_{1_0} + c[0] + c[1]$$
$$s_{2_2} = s_{2_1} + s_{1_2} = s_{2_0} + s_{1_0} + c[0] + s_{1_1} + c[1] = s_{2_0} + s_{1_0} + c[0] + c[1] + s_{1_0} + c[0]$$
$$= s_{2_0} + 2\,s_{1_0} + 2\,c[0] + c[1]$$
$$\dots$$
$$s_{1_N} = s_{1_{(N-1)}} + c[N] = s_{1_0} + \sum_{i=1}^{N} c[i]$$
$$s_{2_N} = s_{2_0} + N s_{1_0} + \sum_{i=1}^{N} (N - i + 1)\,c[i]$$

And since Altivec processes 16 bytes at a time, we can use N = 16:

$$s_{1_{16}} = s_{1_0} + \sum_{i=1}^{16} c[i]$$
$$s_{2_{16}} = s_{2_0} + 16\,s_{1_0} + \sum_{i=1}^{16} (16 - i + 1)\,c[i]$$

In this formulas, `s1_0` and `s2_0` hold the values of `s1` and `s2` at the beginning of the calculation. Now, those of you with Altivec experience will immediately recognize that both of these quantities can easily be calculated using `vec_msum` and `vec_sums`. Specifically the loop could look something like this:

```
while (k >= 16) {
    vbuf = (vector unsigned char)vec_ld(0, (unsigned char *)buf);
    vsum1 = (vector signed int)vec_msum(vbuf, v1, v0);
    vs1 = vec_sums(vsum1, vs1);
    vsum2 = (vector signed int)vec_msum(vbuf, v2, v0);
    vs2 = vec_sums(vsum2, vs2);
    buf += 16;
    k -= 16;
    vs1_0 = vec_sll(vs1_0, vsh);
    vs2 = vec_add(vs2, vs1_0);
    vs1_0 = vs1;
}
```

A little explanation is in order: first, it goes without saying that `vbuf` holds the current buffer data as a `vector unsigned char`, and that `vs1`, `vs2` hold the `s1`, `s2` quantities in vector form (`vector signed int`), and `vs1_0` holds the value of `vs1` in the start of the loop. Also, `vsum1` and `vsum2`, are used to hold the sums used in the above formulas. For this reason we define the following constant vectors:

```
v0 = vec_splat_u32(0); // { 0, 0, 0, 0} (32-bit integers)
v1 = vec_splat_u8(1); // ie {  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
v2 = { 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
vsh = vec_splat_u8(4);
```

Now it becomes apparent that the code above is the Altivec implementation of the formulas. Note: Since the term to be added to the sums is `s2_0` and `16*s1_0`, we can just shift 4 bits to the left

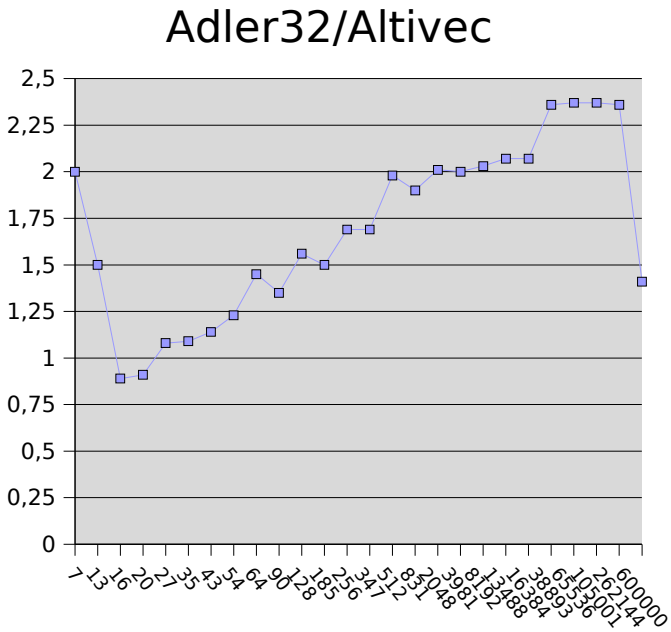
(using `vec_sll`) on `vs1_0`.

At the start of the `NMAX` iterations we have to load the current `s1` and `s2` into vectors. Also at the end of `NMAX` iterations we have to store the values *from* the vectors *to* the scalars, calculate the module and then proceed with the rest `NMAX` iterations. We mentioned that `NMAX` is such so that the inequality is satisfied. But using Altivec there is a slight problem. Both `s1` and `s2` are calculated using the C Altivec operation `vec_sums`, which returns a *signed* int instead of an unsigned. So the `NMAX` calculated for the scalar method has to be recalculated again for the maximum of `2^31 -1`. It is easily found that this number is 3854. Also we have to make sure that the vector processing is done on aligned buffers. So we have to add scalar code at the beginning to take care of the alignment. Scalar code must also be inserted at the end of the stream to handle the remaining few bytes. Given these extra information, we can present now the final Altivec version of the Adler32 algorithm (the code is published as dual (GPL v2/BSD), choose whichever is needed):

```
#include <altivec.h>

#define S1  s1[3]
#define S2  s2[3]
#define NMAX_VEC 3854
/* NMAX_VEC is the largest n such that 255n(n+1)/2 + (n+1)(BASE-1) <= 2^31-1 */

uLong ZEXPORT adler32_vec(adler, buf, len)
    uLong adler;
    const Bytef *buf;
    uInt len;
{
    unsigned long __attribute__ ((aligned(16))) s1[4], s2[4];
    S1 = adler & 0xffff;
    S2 = (adler >> 16) & 0xffff;
    int k, i, offset;

    if (buf == Z_NULL) return 1L;

    // Handle small sizes
    if (len < 16) {
        while (len--) {
            S1 += *buf++;
            S2 += S1;
        }
        MOD(S1);
        MOD(S2);

        return (S2 << 16) | S1;
    }

    if (len >= 16) {
        vector unsigned int v0 = vec_splat_u32(0);
        vector signed int vs1, vs2, vsum1, vsum2, vs1_0;
        vector unsigned char vbuf, v1 = vec_splat_u8(1), v2 = { 16, 15, 14, 13, 12, 11,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1 },
            vsh = vec_splat_u8(4);

        // Align to 16-byte boundaries
        offset = (unsigned long)(buf) % 16;
        if (offset) {
            offset = 16 -offset;
            len -= offset;
            while (offset--) {
                S1 += *buf++;
                S2 += S1;
            }
        }

        while (len >= 16) {
```

```
            vs1 = vec_lde(12, (signed int *)&s1);
            vs2 = vec_lde(12, (signed int *)&s2);
            vs1_0 = vs1;
            k = len < NMAX_VEC ? (int)len : NMAX_VEC;
            k -= k % 16;
            len -= k;
            while (k >= 16) {
                vbuf = (vector unsigned char)vec_ld(0, (unsigned char *)buf);
                vsum1 = (vector signed int)vec_msum(vbuf, v1, v0);
                vs1 = vec_sums(vsum1, vs1);
                vsum2 = (vector signed int)vec_msum(vbuf, v2, v0);
                vs2 = vec_sums(vsum2, vs2);
                buf += 16;
                k -= 16;
                vs1_0 = vec_sll(vs1_0, vsh);
                vs2 = vec_add(vs2, vs1_0);
                vs1_0 = vs1;
            }
            vec_ste(vs1, 12, &s1);
            vec_ste(vs2, 12, &s2);
            MOD(S1);
            MOD(S2);
        }
    }

    while (len--) {
        S1 += *buf++;
        S2 += S1;
    }
    MOD(S1);
    MOD(S2);

    return (S2 << 16) | S1;
}
```

The following chart presents the speed gain of the Altivec version over the original scalar, for aligned data (on 16byte boundaries). Similar results are for unaligned data.



References

The usual Altivec manuals:

The Freescale Altivec PEM/PIM books:
http://www.freescale.com/Altivec/

IBM eServer Bladecenter JS20, PowerPC 970 Programming Environment:
http://www.redbooks.ibm.com/redpapers/pdfs/redp3890.pdf

Apple's Developer Connection, Velocity Engine:
http://developer.apple.com/hardware/ve/index.html

“Altivec Extension to PowerPC Accelerates Media Processing”, Dieffendorf, Dubey, Hochsprung, Scales:
http://www.cse.lehigh.edu/~mschulte/ece401-01/papers/altivec.pdf

And other Altivec-related documents, plus material in mailing lists.