Vectorization of the $h_i = ah_{i-1} + b + x_i$ family of hashing algorithms, using Altivec

June 8, 2005

Abstract

Hashing a piece of data is one of the most time consuming process in a a database. Basically, the process builds a unique integer (aka 'hash key') from a series of operations on the objects bytes. Usually these algorithms are not parallelizable, or at least not easily parallelizable, due to the dependency to previous calculations involved. Taking a single family of hashing algorithms, we will provide (with mathematical proof) a second deterministic way of calculating the Nth hash, using easily parallelizable tecniques and also provide the Altivec equivalent of this algorithm.

1 The $h_i = ah_{i-1} + b + x_i$ family of hashing algorithms

Let's do some explanations first. The values a and b are constants, whereas x_i refers to the current item of the data set, whose hash we want to calculate. The formula is self-explanatory and as we can see, the hash key, which is represented by h_i, is calculated for every element of the data set, but the result also depends on the previous value for h_{i-1}. So if our data set consists of N data items, the final hash will be h_N, but we will need to have calculated all previous values.

It is plainly obvious that a vector unit such as Altivec, or even any parallel infrastructure is useless for this hashing algorithm. The strong dependency on previous elements forbids any optimization efforts using parallel/vector techniques. We have to find another way to do the calculations.

To do that, let's try to calculate the first few elements. For that let's assume that the initial hash key, h_0 is empty, ie. $h_0 = 0$. We'll assume that the element indexing follows the C-like convention, ranging from 0 to $N - 1$.

$$
\begin{aligned}
h_1 &= ah_0 + b + x_0 \\
h_2 &= ah_1 + b + x_1 \\
&= a(ah_0 + b + x_0) + b + x_1 \\
&= a^2 h_0 + (a+1)b + x_1 + ax_0
\end{aligned}
$$

Similarly, let's try to calculate h_3:

$$
\begin{aligned}
h_3 &= ah_2 + b + x_2 \\
&= a(a^2 h_0 + (a+1)b + x_1 + ax_0) + b + x_2 \\
&= a^3 h_0 + a(a+1)b + ax_1 + a^2 x_0 + b + x_2 \\
&= a^3 h_0 + (a^2 + a + 1)b + a^2 x_0 + ax_1 + x_2
\end{aligned}
$$

Anyone see the pattern? For those that don't we suggest that h_N, is of the form:

$$
h_N = a^N h_0 + b \sum_{i=0}^{N-1} a^i + \sum_{i=0}^{N-1} a^{N-i-1} x_i \tag{1}
$$

to confirm that, we'll just have to prove that h_{N+1} follows the same formula:

$$
\begin{aligned}
h_{N-1} &= ah_N + b + x_N \\
&= a\left(a^N h_0 + b \sum_{i=0}^{N-1} a^i + \sum_{i=0}^{N-1} a^{N-i-1} x_i\right) + b + x_N \\
&= a^{N+1} h_0 + ab \sum_{i=0}^{N-1} a^i + a \sum_{i=0}^{N-1} a^{N-i-1} x_i + b + x_N \\
&= a^{N+1} h_0 + b \sum_{i=0}^{N-1} a^{i+1} + \sum_{i=0}^{N-1} a^{N-i} x_i + b + x_N \\
&= a^{N+1} h_0 + b\left(1 + \sum_{i=1}^{N} a^i\right) + \sum_{i=0}^{N-1} a^{N-i} x_i + a^0 x_N \\
&= a^{N+1} h_0 + b \sum_{i=0}^{N} a^i + \sum_{i=0}^{N} a^{N-i} x_i \\
&= a^{N+1} h_0 + b \sum_{i=0}^{(N+1)-1} a^i + \sum_{i=0}^{(N+1)-1} a^{(N+1)-i-1} x_i
\end{aligned}
$$

which has exactly the form of eq. 1. So, we have proven an alternative form to calculate h_N, and we can immediately deduce the following conclusions:

- The two first terms can be considered as constant. Indeed, the only dependency on the actual data is in the third term.
- Equation 1 can be used in steps, eg. first calculate h_N, and then use h_N as h_0 to calculate h_{2N}, etc.

Now, we only have to implement eq. 1 in software using vector SIMD techniques and in particular Altivec, which is our SIMD technology of choice.

2 Implementing the new algorithm in Altivec

Before we start writing code we have to make a few observations:

- We note that our data set consists of bytes, which means that all x_i elements are 8-bit values.
- Most hashing algorithms produce 32-bit unsigned integers as hashes. For this reason, we will assume that all constants are 32-bit integers and that all operations will be performed with 32-bit integer arithmetic, unsigned.
- As we said the first and 2nd term are constants, so, for performance reasons, these will be precalculated and replaced by other constants.
- Since most SIMD platforms operate on 128-bit registers (ie 16 bytes), we will do our hashing calculations on 16 bytes at a time. This means, we will compute first h_{16}, then h_{32}, etc.

Taking these observations into considerations, we might rewrite eq 1 as following:

$$h_{16} = Ah_0 + B + S_{15} \tag{2}$$

where (for $N = 16$),

$$\begin{aligned} A &= a^{16} \\ B &= b\sum_{i=0}^{15} a^i \\ C_i &= a^{15-i} \\ S_{15} &= \sum_{i=0}^{15} C_i x_i \end{aligned}$$

Quantities A, B and C_i are of course constants, ie they do not depend on x_i. So, we can now start implementing eq. 1 in code? Not quite. We have stated that we will be using 32-bit integers, but Altivec does not support 32-bit multiplication, and we plainly see that eq. 1 uses multiplication. According to [**?**], 32-bit multiplication can be implemented with a trick, and we will use

a variation of that for our purposes. We will split the 32-bit quantity in 2 16-bit quantities and multiply each half and add them together afterwards, also doing the appropriate shifting on the most significant half. So, in essence what we will do is:

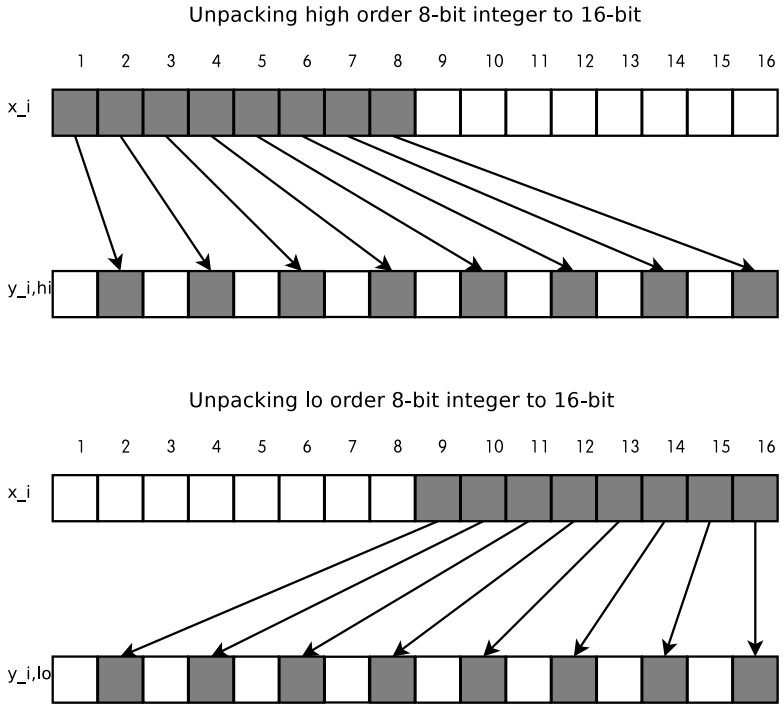
$$\begin{aligned} S_{15} &= \sum_{i=0}^{15} C_i x_i \\ &= \sum_{i=0}^{15} (C_{i,lo} + C_{i,hi} << 16)\, x_i \\ &= \sum_{i=0}^{15} C_{i,lo} x_i + \sum_{i=0}^{15} (C_{i,hi} << 16)\, x_i \\ &= \sum_{i=0}^{15} C_{i,lo} x_i + \left(\sum_{i=0}^{15} C_{i,hi} x_i \right) << 16 \end{aligned}$$

Note that, while C_i were 32-bit unsigned integer quantities, $C_{i,lo}$ and $C_{i,hi}$ are 16-bit unsigned integer quantities, and we have to left-shift a 16-bit integer 16 bits in order to make it a 32-bit integer. Also note that left-shifting is, in essence, just multiplication so we can treat it as such and move it out of the summation and do just one final shift on the 16-bit integer sum instead of shifting every 16-bit term in the sum.

Now, we also have to make sure that the multiplications are done between same size elements. As it is, Altivec does not allow multiplications of 8-bit vs 16-bits or 32-bit elements. Since $C_{i,lo}$ and $C_{i,hi}$ are 16-bit unsigned integers, we have to convert x_i to 16-bit quantities as well (we have considered them as 8-bit unsigned integers). We can do that by 'unpacking' x_i. Ideally we would use 'unpackh' to convert the 8 most significant bytes of the vector to 16-bit integers and 'unpackl' for the 8 least significant bytes of the vector. But unfortunately Altivec does not support unpacking of unsigned quantities (x_i are unsigned 8-bit chars). Before we do the unpacking we split again the sums:

$$\begin{aligned} S_{15} &= \sum_{i=0}^{7} C_{i,lo} x_i + \sum_{i=8}^{15} C_{i,lo} x_i \\ &+ \left(\sum_{i=0}^{7} C_{i,hi} x_i \right) << 16 + \left(\sum_{i=8}^{15} C_{i,hi} x_i \right) << 16 \end{aligned}$$

Unpacking x_i to two new arrays $y_{i,lo}$ and $y_{i,hi}$ would mean doing the following:



The arrays $y_{i,lo}$ and $y_{i,hi}$ would use a different indexing scheme, though, both using a range $[0,7]$. In order to have consistency in the sums, we instead create a 2-dimensional array D_{ij} from C_i, so that:

$$D_{ij} = C_{8i+j}$$

So using these results, we have the final form:

$$\begin{aligned} S_{15} &= \sum_{i=0}^{7} D_{0,i,lo} y_{i,hi} + \sum_{i=0}^{7} D_{1,i,lo} y_{i,lo} \\ &+ \left(\sum_{i=0}^{7} D_{0,i,hi} y_{i,hi} \right) << 16 + \left(\sum_{i=0}^{7} D_{1,i,hi} y_{i,lo} \right) << 16 \end{aligned}$$

So we must calculate each term in S_{15} and then add them all together to calculate h_{16} in the end. We add the following definitions:

$$
\begin{aligned}
s_1 &= \sum_{i=0}^{7} D_{0,i,lo} y_{i,hi} \\
s_2 &= \sum_{i=0}^{7} D_{1,i,lo} y_{i,lo} \\
s_3 &= \sum_{i=0}^{7} D_{0,i,hi} y_{i,hi} \\
s_4 &= \sum_{i=0}^{7} D_{1,i,hi} y_{i,lo}
\end{aligned}
$$

We must not forget that we have to shift left 16-bits the sums s_3 and s_4 after we calculate them. So now can we start writing code? Yes, we'll start from the unpacking issue. As we said earlier, Altivec doesn't support unpacking unsigned entities, so we have to use a the `vec_mergeh`/`vec_mergel` instructions. Assuming we have loaded 16 bytes of the buffer to the vector of 8-bit unsigned char elements `vx` and we want to create the two unpacked vectors of 16-bit unsigned short elements `vy_hi` and `vy_lo`:

```
vector unsigned char vx, zero = vec_splat_u8(0);
vector unsigned short vx;
vx = vec_ld(0, buf);
vy_hi = (vector unsigned short) vec_mergeh(zero, vx);
vy_lo = (vector unsigned short) vec_mergel(zero, vx);
```

We had to use casting in the calls to `vec_mergeh`/`vec_mergel` as these return `unsigned char` vectors. Now let's calculate s_1, s_2, s_3 and s_4, but before that let's define the constants D_{ij}:

```
vector unsigned short D_lo[2], D_hi[2];
```

These arrays will hold the precalculated values of the D_{ij}, which are in essense the high and low 16-bits of the 32-bit integers that represent the powers of a.

```
vector unsigned int vs1, vs2, vs3, vs4;
vs1 = vec_msum( D_lo[0], vy_hi, zero);
vs2 = vec_msum( D_lo[1], vy_lo, vs1);
vs3 = vec_msum( D_hi[0], vy_hi, zero);
vs4 = vec_msum( D_hi[1], vy_lo, vs3);
```

The resulting vectors hold the results but in 4 32-bit quantities in each vector. We should add these together, but before that, let's first do the shifting in the s_4 vector:

```
vector unsigned int v5, shift16 = vec_splat_u32(16);
vs5 = vec_sl( vs4, shift16);
```

Now add all the vectors together:

```
vector unsigned int vs6;
vs6 = vec_add( vs2, vs4);
```

The new vector s_5 holds the final 4 32-bit quantities and we need to add these together to get the final S_{15} sum. You might notice that we can combine the one `vec_msum()` and the `vec_add()` commands into one. So instead we might use this piece of code (we have renamed the vector registers as well):

```
vs1 = vec_msum( D_hi_0, vy1_hi, zero );
vs2 = vec_msum( D_hi_1, vy1_lo, vs1 );
vs3 = vec_sl( vs2, shift16 );

vs4 = vec_msum( D_lo_0, vy1_hi, vs3);
vs5 = vec_msum( D_lo_1, vy1_lo, vs4 );
```

Unfortunately, Altivec does not offer such an instruction (actually it does, `vec_msum`, but this one supports only signed quantities. So the easiest and fastest way is to store the resulting s_5 vector and carry off from there with plain scalar integer addition. We do the store on a 16-byte aligned integer array.

```
unsigned int __attribute__((aligned(16))) s15[4];
unsigned int sum15;
vec_st(vs5, 0, &s15);
sum15 = s15[0] + s15[1] + s15[2] + s15[3];
```

Now according to 2 we just have to add the constants Ah_0 and B to get the final hash h for these 16 bytes.

```
unsigned char a[17], basum[15];
h = a[16]*h0 + basum[15] + sum15;
```

By the way, in case you haven't already guessed, the arrays *a* and *basum* hold the 17 and 16 elements of A and B. Why 17? Remember that C uses the range $(0, N-1)$ for an array of N elements, so since we use $a[16]$ in our calculations, the array has to have 17 elements.

So, is this over? No, we still have to provide a final step, to ensure that the algorithm works correctly in each iteration, since we're going to use it in steps of 16-bytes. So at the end of each iteration, we have to set $h0$ equal to the current h.

```
h0 = h;
```

So let's put everything together:

```
unsigned int __attribute__((aligned(16))) s15[4];
unsigned int sum15;
unsigned char a[17], basum[15];

vector unsigned char vx;
vector unsigned short vy_hi, vy_lo;
vector unsigned short D_lo[2], D_hi[2];
vector unsigned int vs1, vs2, vs3, vs4, vs5;
vector unsigned int shift16 = vec_splat_u32(16);
vector unsigned int zero = vec_splat_u32(0);

while (len >= 16) {
    vx = vec_ld(0, buf);
    vy_hi = (vector unsigned short) vec_mergeh(zero, vx);
    vy_lo = (vector unsigned short) vec_mergel(zero, vx);

    vs1 = vec_msum( D_hi[0], vy1_hi, zero );
    vs2 = vec_msum( D_hi[1], vy1_lo, vs1 );
    vs3 = vec_sl( vs2, shift16 );

    vs4 = vec_msum( D_lo[0], vy1_hi, vs3);
    vs5 = vec_msum( D_lo[1], vy1_lo, vs4 );

    vec_st(vs5, 0, &s15);
    sum15 = s15[0] + s15[1] + s15[2] + s15[3];
    h = a[16]*h0 + basum[15] + sum15;
    buf += 16;
    len -= 16;
    h0 = h;
}
```

Of course, as you can see for yourselves, the whole Altivec loop has to happen only if the length of the buffer is large enough (*len* >= 16).

3 Catering for the scalar cases

So is this over? No, for the routine to be complete we have to cater for the first few bytes (until `buf` is 16-byte aligned), and the last remaining bytes (less than 16).

So one easy way to do that is to add the following code before the Altivec loop:

```
unsigned char c;
int offset = (unsigned int) buf % 16;
if (offset) {
```

```
    h = basum[offset];
    len -= offset;
}
while (offset--) {
    c = *buf++;
    h += a[--len]*c;
}
h0 = h;
```

And the following after the Altivec loop:

```
if (len) {
    h = a[len]*h0 + basum[len-1];
    for (i = len; i > 0; i--) {
        c = *buf++;
        h += a[i-1]*c;
    }
}
```

As you will probably notice, if you compare this code with the actual code in the original hashing functions (we have used the hashing functions in the Berkeley DB `hash_func.c` as basis), it will probably not be very fast. In fact, this will be quite slow, compared to the manually loop-unrolled hand optimized code there. Benchmarks have shown that for small sizes (where Altivec matters little) this code is about 50% the speed of the original scalar code. Clearly we have to optimize this as well. One easy way is to use a mix of the Altivec and the unrolling techniques and use them for our purpose. So the end code might be transformed into the following:

```
if (len >= 8) {
    vector uint16_t D_hi;
    vector uint16_t D_lo;

    vector uint32_t vs1, vs2, vs3, vs4;
    vector uint8_t vx;
    vector uint16_t vy;
    vector uint32_t shift16 = vec_splat_u32(-16);
    vector uint32_t zero = vec_splat_u32(0);
    vx = vec_ld(0, k);
    vy = (vector uint16_t) vec_mergeh((vector uint8_t)zero, vx);

    vs1 = vec_msum( D_hi, vy, zero );
    vs2 = vec_sl( vs1, shift16 );
    vs3 = vec_msum( D_lo, vy, vs2 );

    vec_st(vs3, 0, &s15[0]);
    sum15 = s15[0] + s15[1] + s15[2] + s15[3];
```

```
    h = a[8]*h0 + basum[7] + sum15;

    k += 8;
    len -= 8;
    h0 = h;
}

if (len) {
    h = a[len]*h0 + basum[len-1];
    switch (len) {
        case 7:
            h += a[6]*k[0] + a[5]*k[1] +a[4]*k[2] + a[3]*k[3]
                 + a[2]*k[4] + a[1]*k[5] + a[0]*k[6];
            break;
        case 6:
            h += a[5]*k[0] + a[4]*k[1] +a[3]*k[2] + a[2]*k[3]
                 + a[1]*k[4] + a[0]*k[5];
            break;
        case 5:
            h += a[4]*k[0] + a[3]*k[1] +a[2]*k[2] + a[1]*k[3]
                + a[0]*k[4];
            break;
        case 4:
            h += a[3]*k[0] + a[2]*k[1] +a[1]*k[2] + a[0]*k[3];
            break;
        case 3:
            h += a[2]*k[0] + a[1]*k[1] +a[0]*k[2];
            break;
        case 2:
            h += a[1]*k[0] + a[0]*k[1];
            break;
        case 1:
            h += a[0]*k[0];
            break;
    }
}
```

We might even optimize the Altivec code a bit more. Tests have shown that we might get a bit more if we unroll the Altivec loop twice (so that it handles 32 bytes at once). The final and complete Altivec hashing routine should be accompanying this document.

4 Example algorithms

Let's test this algorithm against some known ones. We have chosen 3 algorithms that fall in this category and which are included in the Berkeley DB, in the file `hash_func.c`. The respective functions are called `__ham_func2()`, `__ham_func3` and `__ham_func4` (included in the accompanying `hashing.c`).

If we take a closer look we'll immediately see that these algorithms are of the form $h_i = ah_{i-1} + b + x_i$. Trully, for the first one, it's immediately apparent, as we can see:

```
(h) = 0x63c63cd9*(h) + 0x9c39c33d + (c)
```

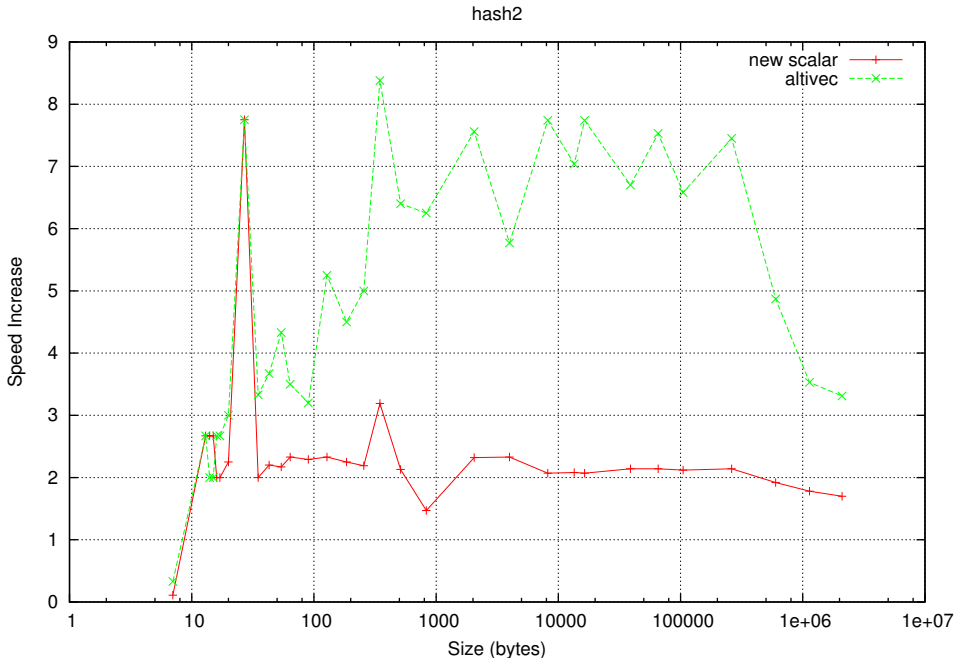
So $a = 0x63c63cd9$ and $b = 0x9c39c33d$. For the second and third algorithms, we can see that $b = 0$, and that $a = 65559$ for the second and $a = 33$ for the 3rd:

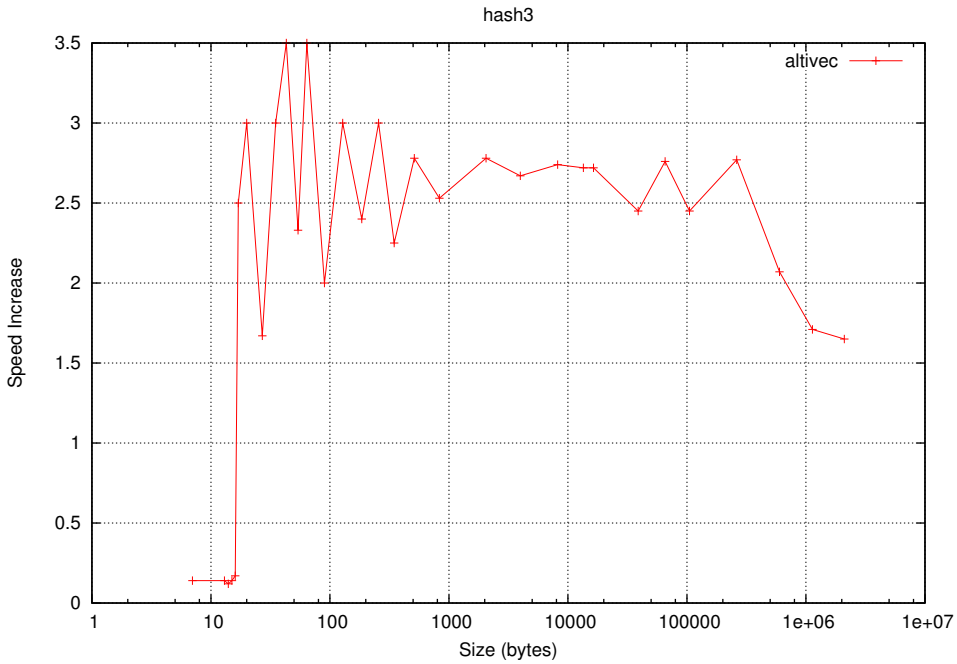
```
n = *k++ + 65599 * n
```

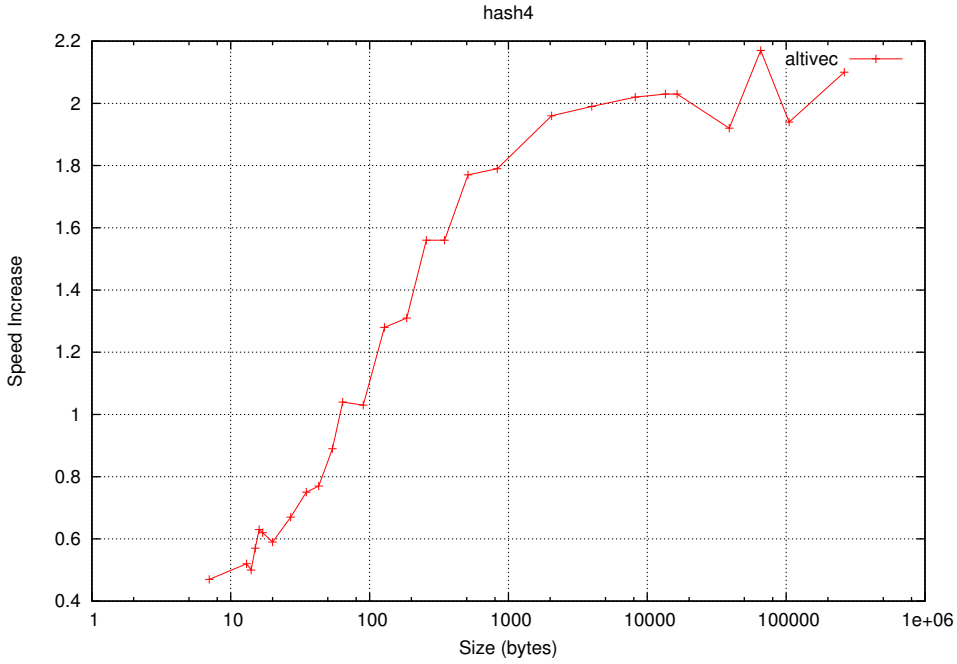
and

```
h = (h << 5) + h + *k++ = h * 32 + h + *k++ = 33*h + *k++
```

So, the only remaining step is to calculate the array $a[17]$, $basum[16]$ and $D_hi[2][8]$, $D_lo[2][8]$. These are easily computed and you can see the results in each function in the attached source. We now only have to provide performance results of each vectorized algorithm against the original function.




hash3
altivec
Speed Increase
Size (bytes)
3.5
3
2.5
2
1.5
1
0.5
0
1
10
100
1000
10000
100000
1e+06
1e+07


hash4
altivec
Speed Increase
Size (bytes)
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
1
10
100
1000
10000
100000
1e+06