Improvevement of Fitch function for Maximum Parsimony in Phylogenetic Reconstruction with Intel AVX2 assembler instructions

Jean-Michel Richer

université angers
FACULTÉ
DES SCIENCES
Unité de formation et de recherche

Technical Report

Improvement of Fitch function for Maximum Parsimony in Phylogenetic Reconstruction with Intel AVX2 assembler instructions

Research Lab: LERIA

TR20130624-1
Version 1.0
24 June 2013

JEAN-MICHEL RICHER
Office: H206
Address: 2 Boulevard Lavoisier, 49045 Angers Cedex 01
Phone: (+33) (0)2-41-73-52-34
Email: jean-michel.richer@univ-angers.fr

Abstract

The Maximum Parsimony problem aims at reconstructing a phylogenetic tree from DNA, RNA or protein sequences while minimizing the number of evolutionary changes. Much work has been devoted by the Computer Science community to solve this NP-complete problem and many techniques have been used or designed in order to decrease the computation time necessary to obtain an acceptable solution. In this paper we report an improvement of the evaluation of the Fitch function for Maximum Parsimony using AVX2 assembler instruction of Intel^{TM} processors.

1 Introduction

This report is an extension of the technical report **TR20080428-1** by the same author. We give here a new version of the assembler code and we have performed some tests on an Intel Haswell processor to verify if the AVX2 assembler instruction set gave any improvement over SSE2. For more details we refer the reader to the technical report **TR20080428-1**.

1.1 Software improvement using AVX2 instructions

The release of the new Haswell architecture of Intel processor in June 2013 led to the introduction of **AVX2** (*Advanced Vector Extensions, version 2*) assembler instructions. With AVX, introduced in 2008, the width of the SIMD registers is increased from 128 bits to 256 bits and the SSE registers `xmm0-xmm15` are renamed to `ymm0-ymm15` for a 64 bits architecture.

AVX2 extensions like **SSE2** (*Streaming SIMD Extensions*) instructions of modern x86 processors (Intel, AMD) help vectorize the code, i.e. apply the same instruction on multiple data at the same time consequently reducing the overall execution time.

In our implementation of phylogenetic reconstruction with Maximum Parsimony using Fitch criterion, the main function that benefits from the use of vectorization is the computation of a hypothetical parsimony sequence from two existing sequences. The C code of this function is given figure 1 and takes as input two sequences `x` and `y` of a given `size`. The outputs are the hypothetical taxon `z` and the number of `changes` (or differences) returned by the function.

```
1 int fitch(char x[], char y[], char z[], int size) {
2   int i, changes=0;
3   for (i = 0; i < length; ++i) {
4     z[i] = x[i] & y[i];
5     if (z[i] == 0) {
6       ++changes;
7       z[i] = x[i] | y[i];
8     }
9   }
10  return changes;
11 }
```

Figure 1: Fitch Parsimony function

Modern compiler (gcc GNU, icc Intel) are not able to vectorize the code of this function efficiently if no implementation specific information is provided. It is then necessary to code the function in assembler to get a significant improvement during the execution of the program.

The implementation with AVX2 is nearly the same as the one given in report TR20080428-1 for SSE2:

1. we first load into registers $ymm0$ and $ymm1$ the first 32 bytes of each taxon (`x` and `y`)
2. in $ymm2$ and $ymm3$, we respectively compute the binary-AND and the binary-OR of $ymm0$ and $ymm1$ using instructions `vpand` and `vpor` (for parallel AND and parallel OR).

3. then, we compare $ymm2$ with a vector of zero using the instruction `vpcmpeqb` (which performs a parallel comparison of each byte of two ymm registers) in order to determine which bytes of $ymm3$ will replace the zero values of $ymm2$. The result is stored in register $ymm5$. The result of `vpcmeqb` is such that $ymm5[i] = 0$, if originally $ymm2[i] = 0$ otherwise $ymm5[i] = 255$

4. as a consequence, we can use $ymm5$ and combine it with $ymm2$ and $ymm3$ to get the final result of taxon `z` by calculating : $(\mathrm{ymm5} \,\&\, \mathrm{ymm3}) | (NOT(\mathrm{ymm5}) \,\&\, \mathrm{ymm2})$. This process is repeated every 32 bytes until we reach the last bytes of the taxa.

5. when the size of the taxa is not a multiple of 32, the last part of the taxon `z` is computed using a traditional implementation which treats one byte at a time.

The number of changes is evaluated using the POPCNT (for POPulation CouNT) instruction which counts the number of bits set to 1 in a general purpose register. Note also that we take advantage of AVX2 by using the `vpcmpeqb` on a ymm register which compares in parallel the 32 bytes of two ymm registers.

An overview of the assembler code is given figure 2:

```
 1 pxor          ymm4, ymm4          ; initialize ymm4 with 0 for comparison
 2 vmovdqa       ymm0, [ebx]         ; load x[ebx+0:31] into ymm0
 3 vmovdqa       ymm1, [esi]         ; load y[esi+0:31] into ymm1
 4 vpand         ymm2, ymm0, ymm1    ; ymm2 <- ymm0 & ymm1
 5 vpor          ymm3, ymm0, ymm1    ; ymm3 <- ymm0 | ymm1
 6 vpcmpeqb      ymm5, ymm4, ymm2    ; compare ymm2 to ymm4 (AVX2)
 7 vpmovmskb     edx, ymm5           ; store in ymm5 and put number of bits in edx
 8 popcnt        edx, edx            ; use popcnt to compute number of bits
 9 add           eax, edx            ; add to eax which records number of changes
10 vpblendvb     ymm0, ymm2, ymm3, ymm5 ; compute result
11 vmovdqa       [edi], ymm0         ; store in z[edi+0:31]
```

Figure 2: Translation of the Fitch parsimony function (see fig. 1) in assembler x86 32 bits using AVX2 instructions.

In our implementation we use a special instruction called `vpblendvb` which is equivalent to three instructions (see figure 3). We also use *loop unrolling* of 4, which means that we can unroll the loop 4 times in order to treat 4×32 elements in one iteration.

```
1 vpand         ymm0, ymm3, ymm5    ; ymm0 <- (ymm0 | ymm1) & ymm5
2 vpandn        ymm1, ymm5, ymm2    ; ymm1 <- (ymm0 & ymm1) & ~ymm5
3 vpor          ymm0, ymm0, ymm1    ; ymm0 <- ymm0 | ymm1
```

Figure 3: Equivalence of the `vpblendvb` instruction

2 Benchmark and results

To assess the performance of our AVX2 implementation we have designed a simple benchmark. The benchmark consists in a loop executed 200.000 times on a set of 100 sequences for different sizes (127, 255, ..., 4095). We apply the Fitch function between the sequences. Sequences are dynamically allocated using `_mm_malloc` and are aligned on a 32 bytes boundary, i.e. the addresses of the memory blocks allocated are a multiple of 32. The benchmark was compiled and run under Ubuntu 13.04 32 and 64 bits versions.

2.1 Results On Intel i5 4570 Haswell

We have implemented the AVX2 version of the Fitch function and compared it to the SSE2 version released in 2008. Results were obtained on an Intel Core^{TM} i5 4570 CPU running at 3.20GHz that imple-

ments AVX2. Table 1 compares the basic C implementation compiled with gcc (`-O2 -funroll-loops --param max-unroll-times=8 -ftree-vectorize -msse2 -ftree-loop-optimize`) to the assembler SSE2 and AVX2 implementations on a 32 bits architecture. Table 2 reports results for a 64 bits architecture. All the SSE2 and AVX2 implementation use the POPCNT function and have been compiled using NASM 2.10.07, the Netwide Assembler (`http://www.nasm.us`).

In both cases, columns %sse2 and %avx2 reports the improvement in percentage compared to the C implementation for different size of taxa that range from 127 to 4095 residues.

size	C (s)	sse2 (s)	%sse2	avx2 (s)	%avx2
127	2.620	0.800	69.47%	**0.760**	70.99%
255	7.740	0.970	87.47%	**0.850**	89.02%
511	20.900	1.320	93.68%	**1.040**	95.02%
1023	47.060	2.050	95.64%	**1.390**	97.05%
2047	98.330	3.650	96.29%	**2.140**	97.82%
4095	198.330	6.520	96.71%	**3.920**	98.02%

Table 1: Results on a 32 bits architecture of the execution of the benchmark with C, SSE2 and AVX2 implementations for a Core i5 4570 and compiled with gcc 4.7

size	C	sse2 (s)	%sse2	avx2 (s)	%avx2
127	1.650	0.650	60.61%	**0.580**	64.85%
255	6.080	0.810	86.68%	**0.650**	89.31%
511	18.500	1.160	93.73%	**0.850**	95.41%
1023	43.120	1.890	95.62%	**1.260**	97.08%
2047	90.630	3.420	96.23%	**1.950**	97.85%
4095	184.000	6.420	96.51%	**3.750**	97.96%

Table 2: Results on a 64 bits architecture of the execution of the benchmark with C, SSE2 and AVX2 implementations for a Core i5 4570 and compiled with gcc 4.7

From the results obtained we can make the following comments :

- for the C, SSE2, AVX2 implementations, the 64 bits version is faster than the 32 bits version,
- the AVX2 assembler version is the fastest compared to C and SSE2 implementation

2.2 Result on Intel i5 3570 Ivy Bridge

The Intel Core^{TM} i5 3570k CPU @ 3.40GHz does not implement AVX2, but only AVX, so we have decided to compare the results of the C and SSE2 implementations.

size	C	sse2	%sse2
127	3.840	0.920	76.04%
255	10.780	1.120	89.61%
511	22.790	1.540	93.04%
1023	48.390	2.400	95.04%
2047	99.910	4.080	95.92%
4095	201.750	7.260	96.40%

Table 3: Results on a 32 bits architecture of the execution of the benchmark with C and SSE2 implementations for a Core i5 3570k and compiled with gcc 4.7

Although the 3570k is running at a higher frequency than the 4570, we can see that it takes more time to the 3570k to complete the benchmark.

More results for different processor architectures can be found on this page: `http://www.info.univ-angers.fr/pub/richer/ensl3i_crs6.php#parcimonie`.

2.3 Use of Intel compiler on Haswell

2.3.1 vectorization

The Fitch function on figure 2 can be compiled quite efficiently with icpc, the Intel C++ compiler. We used the version 13.1.1 of icpc which is compatible with gcc version 4.7.0. For example by using `assume` directives we can tell the compiler that data are aligned on a 32 byte memory boundary in order for the compiler to use ymm registers.

```
1 #include <stdint.h>
2
3 uint32_t parsimony_1(uint8_t *x, uint8_t *y,
4     uint8_t *z, uint32_t size) {
5 __assume_aligned(x, 32);
6 __assume_aligned(y, 32);
7 __assume_aligned(z, 32);
8
9   #pragma simd
10  for (i=0; i<size; ++i) {
11    // code of Fitch function
12  }
13 }
```

Figure 4: Fitch Parsimony function with Intel icpc directives

The function is compiled with `-O3 -xCORE-AVX2` flags. The `pragma` directive can also be used to provide implementation-defined information to the compiler: the pragma simd directive tells the compiler to enforce vectorization of loops. The execution times show an important improvement of the C implementation but for small sizes (see table 4, column icpc1). We could also use the pragma directive with `simd vectorlength(32)` to tell that we want to treat 32 bytes at a time. This information decreases the execution time (column icpc2 of table 4) compared to icpc1 for a size of vector greater than 1023.

size	icpc1 (s)	icpc2 (s)	avx2 (s)
127	**0.550**	1.200	0.770
255	0.860	1.340	**0.850**
511	1.430	2.040	**1.030**
1023	2.520	2.920	**1.400**
2047	4.640	4.620	**2.140**
4095	8.940	7.960	**3.890**

Table 4: Results on a 32 bits architecture of a Core i5 4570 when compiled with icpc and vectorization directives

2.3.2 profile-guided optimization

PGO (for *Profile-Guided Optimization*) tells the compiler which areas of an application are most frequently executed. The compiler is then able to use feedback from a previous compilation to be more selective in optimizing the application. PGO with Intel icpc gave interesting results (see table 5) for the code of figure 2:

size	pgo (s)	avx2 (s)
127	**0.790**	0.820
255	1.010	**0.920**
511	1.500	**1.170**
1023	2.520	**1.460**
2047	4.400	**2.240**
4095	8.030	**3.940**

Table 5: Results on a 32 bits architecture of a Core i5 4570 when compiled with icpc and PGO directives

2.4 Use of GNU C compiler on Haswell

It was not possible to vectorize the code with gcc 4.7.0 although I have used some directives related to auto vectorization of the code. So I have decided to use *intrinsics* functions which are functions available for use in a given programming language whose implementation is handled specially by the compiler. The most efficient function uses AVX2 intrinsics (see figure 5).

```
1  uint32_t parsimony_intrinsics3(uint8_t *x, uint8_t *y,
2    uint8_t *z, uint32_t size) {
3    uint32_t i, changes=0;
4
5    __m256i _x, _y, _x_and_y, _x_or_y, _zero, _cmp;
6
7    _zero = _mm256_set_epi8(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
8      ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
9
10   for (i = 0; i < (size & (~31)); i+=32) {
11     _x = _mm256_load_si256((__m256i *) &x[i]);
12     _y = _mm256_load_si256((__m256i *) &y[i]);
13     _x_and_y = _mm256_and_si256(_x, _y);
14     _x_or_y = _mm256_or_si256(_x, _y);
15     _cmp = _mm256_cmpeq_epi8(_zero, _x_and_y);
16     uint32_t r = _mm256_movemask_epi8(_cmp);
17     changes += _mm_popcnt_u32(r);
18     _x = _mm256_blendv_epi8(_x_and_y, _x_or_y, _cmp);
19     _mm256_store_si256( (__m256i *) &z[i], _x);
20   }
21
22   for ( ; i<size; ++i) {
23     z[i] = x[i] & y[i];
24     if (z[i] == 0) {
25       z[i] = x[i] | y[i];
26       ++changes;
27     }
28   }
29
30   return changes;
31 }
```

Figure 5: Fitch Parsimony function with intrinsics

Results are reported on table 6 for a 32 bits architecture and on table 7 for a 64 bits architecture. The AVX2 intrinsics version is close to hand-coded AVX2 version.

size	intrinsics sse2 (s)	intrinsics avx2 (s)	avx2 (s)
127	**0.440**	0.750	0.820
255	**0.590**	0.900	0.860
511	**0.970**	1.130	1.100
1023	1.680	1.580	**1.440**
2047	3.250	2.510	**2.300**
4095	6.190	4.610	**4.010**

Table 6: Results on a 32 bits architecture of a Core i5 4570 when compiled with gcc and use of intrinsics

size	intrinsics sse2 (s)	intrinsics avx2 (s)	avx2 (s)
127	**0.380**	0.500	0.560
255	**0.560**	0.560	0.630
511	**0.950**	0.800	0.810
1023	1.740	1.240	**1.220**
2047	3.330	2.210	**1.960**
4095	6.960	4.670	**3.960**

Table 7: Results on a 64 bits architecture of a Core i5 4570 when compiled with gcc and use of intrinsics

3 Conclusion

In this article we have introduced an implementation improvement which relies on the capabilities of modern processors to vectorize data treatments. We think that it would be worth to implement those techniques into existing softwares that try to solve Maximum Parsimony with the Fitch criterion. However the AVX2 instruction set does not improve significantly the execution time compared to its SSE2 version, but the assembler hand-coded versions of the Fitch function are generally more efficient than the ones generated by compilers especially for sequences of size greater than 1023.