HPC - Projet n°1 : Optimisations d'un outil open source

Author: Léonard Favre

Introduction

Choix du logiciel et motivations

Le logiciel que j'ai choisi pour ce projet est SoX, acronyme de SOund eXchange, un utilitaire en ligne de commande permettant des conversions de fichiers audio. En suivant les conseils du professeur et de son assistant, je vais me concentrer sur les effets ne se basant pas sur des librairie afin d'optimiser un d'entre eux. La version sur laquelle l'optimisation se portera est la version 14.4.2.

Procédure d'installation

Le code source de notre logiciel peut se trouver à l'adresse suivante :

https://sourceforge.net/projects/sox/files/sox/

Un fois les dossiers télécharger, se rendre dans le dossier `sox-14.4.2`, puis lancer les commandes suivantes :

```
./configure
make -s
make install
```

Plus d'info concernant l'installation peut être trouvé dans le fichier `INSTALL`

Pour lancer l'effet que j'ai choisi d'optimiser, le chorus, il faut lancer la commande suivante, depuis le dossier src :

`./sox <input> <output> chorus <gain-in> <gain-out> <delay> <decay> <speed> <depth> [ -s | -t ]`

exemple :

`./sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.0 1.0 -s`

Benchmark de base et stratégie

Pour commencer l'analyse, nous lançons une mesure `time` afin d'avoir une idée du temps d'exécution du chorus sur un fichier de test.

```
$ time ./sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.0 1.0 -s

real    0m0.820s
user    0m0.712s
sys     0m0.110s
```

Le but de ce projet sera de faire baisser ces valeurs, rendant ainsi le programme plus rapide et plus performant.

Avec l'aide de la commande `strace`, observons les appels systèmes effectués par notre chorus avec le même exemple :

```
$ strace -c ./sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.0 1.0 -s
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 39.10    0.017968           4      4414           write
 29.96    0.013767           1      8849           futex
 14.90    0.006845         220        31        12 openat
 10.10    0.004641           2      2241           read
  1.84    0.000846          70        12         6 wait4
  0.65    0.000300           6        50           mmap
  0.40    0.000183          91         2           execve
  0.39    0.000180           2        64           rt_sigprocmask
  0.39    0.000178           6        29         8 stat
  0.34    0.000155          25         6           clone
  0.31    0.000143           3        40           rt_sigaction
  0.29    0.000131           4        29           close
  0.18    0.000082          20         4           munmap
  0.18    0.000081           5        15           mprotect
  0.16    0.000074           5        14           pread64
  0.16    0.000073           3        20           fstat
  0.14    0.000066           5        12         2 ioctl
  0.06    0.000029           2        10           lseek
  0.06    0.000029           3         8           brk
  0.06    0.000027           5         5           pipe
  0.04    0.000019           4         4         3 access
  0.04    0.000017           2         6           rt_sigreturn
  0.03    0.000016           8         2           getpid
  0.03    0.000014           4         3         1 fcntl
  0.02    0.000011           3         3           prlimit64
  0.02    0.000010           2         4         2 arch_prctl
  0.02    0.000010           5         2           getdents64
  0.02    0.000009           9         1           sysinfo
  0.01    0.000006           6         1           dup2
  0.01    0.000006           6         1           uname
  0.01    0.000005           5         1           getppid
  0.01    0.000005           5         1           getpgrp
  0.01    0.000004           4         1           getuid
  0.01    0.000004           4         1           getgid
  0.01    0.000004           4         1           geteuid
  0.01    0.000004           4         1           getegid
  0.01    0.000003           3         1           sched_getaffinity
  0.00    0.000002           2         1           set_tid_address
  0.00    0.000002           2         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    0.045949                 15891        34 total
```

On peut déjà voir ici que notre filtre utilise presque 40% de son temps à faire des appels systèmes `write`, ce qui ne parait pas très étonnant pour un programme écrivant un nouveau fichier. On peut voir également que 10% de son temps est consacré à la lecture, avec l'appel système `read` ce qui ne parait pas très étonnant non plus.

On peut observer aussi que 30% du temps est utilisé pour faire des appels systèmes `futex`, qui est l'appel système à la base des mutex et des sémaphores. On peut donc s'attendre à avoir une forme de parallélisation du calcul, cela est normalement bien par rapport au niveau des performances du programme.

Le dernier appel système qui utilise une grande partie du temps de notre programme est `openat`. Il est utile à l'ouverture du fichier d'input.

Nous allons maintenant utiliser le programme PERF afin d'observer les zone du code de notre filtre chorus utilisant le plus de ressources.

Les options qui nous intéressent sont les suivantes :

- cache-misses
- cpu-cycles
- mem-loads
- mem-stores

Nous pouvons faire une première analyse de ces options avec l'option `stat` de PERF :

```
$ sudo perf stat -e cache-misses,cpu-cycles,mem-loads,mem-stores ./sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.
0 1.0 -s

 Performance counter stats for './sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.0 1.0 -s':

         1’875’658      cache-misses
     2’726’925’692      cpu-cycles
                0      mem-loads
       262’337’215      mem-stores

       0.821143515 seconds time elapsed

       0.750923000 seconds user
       0.075350000 seconds sys
```

On voit déjà qu'il n'y a pas de mem-loads, nous pouvons donc abandonné cette option. Les autres valeurs pourrons être utilisées pour comparer les performance de notre amélioration.

Nous allons maintenant collecter ces information et les lié au code source avec l'option `record` de PERF :

```
$ sudo perf record -e cache-misses,cpu-cycles,mem-stores ./sox input/test.wav result/test.wav chorus 1.0 0.6 30.0 0.6 1.0 1.0 -s
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.491 MB perf.data (10180 samples) ]
```

Nous pouvons voir le résultat grâce à l'option `report` de PERF :

```
$ sudo perf report

Available samples
3K cache-misses
3K cpu-cycles
3K mem-stores
```

Nous allons commencer par nous intéresser au cpu-cycles. Le résultat du report nous donne :

```
  53.12%  sox      libsox.so.3.0.0    [.] sox_chorus_flow
  22.33%  sox      libsox.so.3.0.0    [.] flow_no_shape
   4.42%  sox      libsox.so.3.0.0    [.] sox_flow_effects
   1.86%  sox      libsox.so.3.0.0    [.] sox_write_sw_samples
   1.70%  sox      libsox.so.3.0.0    [.] lsx_write_w_buf
   1.44%  sox      libsox.so.3.0.0    [.] sox_read_sw_samples
   1.42%  sox      libsox.so.3.0.0    [.] lsx_read_w_buf
   1.22%  sox      [kernel.kallsyms]  [k] copy_user_enhanced_fast_string
   0.54%  sox      [kernel.kallsyms]  [k] do_syscall_64
   0.39%  sox      [kernel.kallsyms]  [k] clear_page_erms
   0.26%  sox      [kernel.kallsyms]  [k] kmem_cache_alloc
   ...
   ...
```

On peut voir tout de suite que plus de la moiter des cycles du CPU sont utilisées par la fonction `sox_chorus_flow` .

Intéressons nous maintenant aux caches misses, le résultat du report nous donne les informations suivantes :

```
  30.02%  sox      libsox.so.3.0.0    [.] sox_chorus_flow
  27.08%  sox      [kernel.kallsyms]  [k] clear_page_erms
  12.39%  sox      [kernel.kallsyms]  [k] copy_user_enhanced_fast_string
   2.60%  sox      [kernel.kallsyms]  [k] __alloc_pages_nodemask
   1.03%  sox      [kernel.kallsyms]  [k] rmqueue
   1.01%  sox      libsox.so.3.0.0    [.] flow_no_shape
   0.91%  sox      [kernel.kallsyms]  [k] free_pcppages_bulk
   0.83%  sox      libsox.so.3.0.0    [.] sox_flow_effects
   0.71%  sox      [kernel.kallsyms]  [k] prep_new_page
   0.70%  sox      [kernel.kallsyms]  [k] get_page_from_freelist
   0.69%  sox      [kernel.kallsyms]  [k] copy_page
   0.69%  sox      [kernel.kallsyms]  [k] add_to_page_cache_lru
   0.59%  sox      [kernel.kallsyms]  [k] find_get_entries
   0.47%  sox      [kernel.kallsyms]  [k] __add_to_page_cache_locked
   0.46%  sox      [kernel.kallsyms]  [k] pagecache_get_page
   ...
   ...
```

On peut voir que, ici à nouveau, la fonction `sox_chorus_flow` produit le plus de caches misses. On peut remarquer également que la deuxième fonction de notre programme qui produit le plus de cache misses est la même que la deuxième qui demande le plus de cycles de CPU : `flow_no_shape` .

Observons maintenant pour finir l'option `mem-stores` :

```
  36.49%  sox      libsox.so.3.0.0    [.] flow_no_shape
  27.13%  sox      libsox.so.3.0.0    [.] sox_chorus_flow
  12.84%  sox      libsox.so.3.0.0    [.] sox_flow_effects
   6.76%  sox      libsox.so.3.0.0    [.] sox_read_sw_samples
   6.51%  sox      libsox.so.3.0.0    [.] sox_write_sw_samples
   0.20%  sox      [kernel.kallsyms]  [k] _raw_spin_lock_irqsave
   0.19%  sox      [kernel.kallsyms]  [k] do_syscall_64
```

On retrouve à nouveau nos fonctions `flow_no_shape` et `sox_chorus_flow` .

Nous allons maintenant nous concentrer sur la fonction `sox_chorus_flow` .

Analyse en profondeur et idée d'optimisation

Nous allons maintenant observer plus en détail le code de la fonction `sox_chorus_flow` . Toujours grâce à l'outil PERF, nous pouvons observer le code annoté de notre fonction en fonction des critères que l'on a choisi d'observer au chapitre précédent, à savoir :

- cpu-cycles
- cache-misses

En commençant par l'option `cpu-cycle` , une première lecture met en avant les résultat suivant :

```
Percent|
       |
       |
       |     Disassembly of section .text:
       |
       |     000000000001c290 <sox_chorus_flow>:
       |     sox_chorus_flow():
       |     * Processed signed long samples from ibuf to obuf.
       |     * Return number of samples processed.
       |     */
       |     static int sox_chorus_flow(sox_effect_t * effp, const sox_sample_t *i
       |     size_t *isamp, size_t *osamp)
    ...
    ...
       |     chorus->counter - chorus->lookup_tab[i][chorus->phase[i]]) %
       | 80:   mov       0x28(%r9,%rcx,8),%rdx
  0.21 |       mov       0x118(%r9,%rcx,8),%rax
  1.31 |       mov       %r14d,%r15d
       |       sub       (%rax,%rdx,4),%r15d
  0.70 |       mov       %r15d,%eax
  1.06 |       cltd
  0.11 |       idiv      %r13d
 15.46 |       movslq    %edx,%rdx
       |     chorus->maxsamples] * chorus->decay[i];
  1.72 |       movss     (%r10,%rdx,4),%xmm0
  6.91 |       mulss     0x8c(%r9,%rcx,4),%xmm0
  6.58 |       add       $0x1,%rcx
       |     d_out += chorus->chorusbuf[(chorus->maxsamples +
  0.16 |       addss     %xmm0,%xmm1
       |     for ( i = 0; i < chorus->num_chorus; i++ )
  2.79 |       cmp       %ecx,%r8d
       |     ↑ jg        80
       |     /* Adjust the output volume and size to 24 bit */
       |     d_out = d_out * chorus->out_gain;
  0.05 | bb:   mulss     0x6c(%r9),%xmm1
       |     out = SOX_24BIT_CLIP_COUNT((sox_sample_t) d_out, effp->clips);
  6.23 |       cvttss2si %xmm1,%eax
  4.99 |       cmp       $0x7fffff,%eax
       |     ↓ jle       150
       |       addq      $0x1,0xa8(%rdi)
       |       mov       $0x7fffff00,%eax
       |     *obuf++ = out * 256;
  0.16 | dd:   mov       %eax,(%rbx,%r11,4)
    ...
    ...
       |     chorus->phase[i]  =
       |     ( chorus->phase[i] + 1 ) % chorus->length[i];
  0.32 |118:   mov       (%rcx),%rax
  0.65 |       add       $0x1,%rax
  0.15 |       cqto
  0.21 |       idivq     0xb8(%rcx)
 21.88 |       add       $0x8,%rcx
    ...
    ...
```

On peut voir que les instruction utilisant le plus de cycles sont les suivantes :

- `d_out += chorus->chorusbuf[(chorus->maxsamples + chorus->counter - chorus->lookup_tab[i][chorus->phase[i]]) % chorus->maxsample` avec plus de 35%
- `chorus->phase[i] = ( chorus->phase[i] + 1 ) % chorus->length[i];` avec plus de 20%

Continuons maintenant avec l'option `cache-misses`. On obtiens le code assembleur annoté suivant :

```
Percent|
       |
       |
       |     Disassembly of section .text:
       |
       |     000000000001c290 <sox_chorus_flow>:
       |     sox_chorus_flow():
       |     * Processed signed long samples from ibuf to obuf.
       |     * Return number of samples processed.
       |     */
       |     static int sox_chorus_flow(sox_effect_t * effp, const sox_sample_t *i
       |     size_t *isamp, size_t *osamp)
    ...
    ...
       |     d_out += chorus->chorusbuf[(chorus->maxsamples +
       |       mov       0x188(%r9),%r13d
       |       mov       0x20(%r9),%r14d
       |       xor       %ecx,%ecx
  1.01 |       add       %r13d,%r14d
       |       nop
       |     chorus->counter - chorus->lookup_tab[i][chorus->phase[i]]
       |  80:  mov       0x28(%r9,%rcx,8),%rdx
  0.09 |       mov       0x118(%r9,%rcx,8),%rax
  1.24 |       mov       %r14d,%r15d
       |       sub       (%rax,%rdx,4),%r15d
  0.19 |       mov       %r15d,%eax
  1.26 |       cltd
       |       idiv      %r13d
 22.49 |       movslq    %edx,%rdx
       |     chorus->maxsamples] * chorus->decay[i];
  1.12 |       movss     (%r10,%rdx,4),%xmm0
  8.94 |       mulss     0x8c(%r9,%rcx,4),%xmm0
  9.32 |       add       $0x1,%rcx
       |     d_out += chorus->chorusbuf[(chorus->maxsamples +
  0.10 |       addss     %xmm0,%xmm1
       |     for ( i = 0; i < chorus->num_chorus; i++ )
  4.02 |       cmp       %ecx,%r8d
       |     ↑ jg        80
       |     /* Adjust the output volume and size to 24 bit */
       |     d_out = d_out * chorus->out_gain;
       |  bb:  mulss     0x6c(%r9),%xmm1
       |     out = SOX_24BIT_CLIP_COUNT((sox_sample_t) d_out, effp->cl
  6.94 |       cvttss2si %xmm1,%eax
  4.69 |       cmp       $0x7fffff,%eax
       |     ↓ jle       150
    ...
    ...
       |     chorus->phase[i]  =
       |     ( chorus->phase[i] + 1 ) % chorus->length[i];
  0.21 |118:  mov       (%rcx),%rax
  0.64 |       add       $0x1,%rax
  0.09 |       cqto
       |       idivq     0xb8(%rcx)
 14.69 |       add       $0x8,%rcx
    ...
    ...
```

On peut voir que les instruction produisant le plus de cache-miss sont les suivantes :

- `d_out += chorus->chorusbuf[(chorus->maxsamples + chorus->counter - chorus->lookup_tab[i][chorus->phase[i]]) % chorus->maxsample` avec plus de 40%
- `chorus->phase[i] = ( chorus->phase[i] + 1 ) % chorus->length[i];` avec plus de 15%

On retrouve de nouveau les même deux fonctions, cela nous indique qu'une optimisation d'une de ces deux instructions aura un effet plus marqué sur les performances du programme que l'optimisation d'une autre instruction de cette fonction. Nous allons donc nous concentrer sur ces deux instructions.

Le code source de notre fonction `sox_chorus_flow` est le suivant :

```
        while (len--) {
                /* Store delays as 24-bit signed longs */
                d_in = (float) *ibuf++ / 256;
                /* Compute output first */
                d_out = d_in * chorus->in_gain;
                for ( i = 0; i < chorus->num_chorus; i++ )
                        d_out += chorus->chorusbuf[(chorus->maxsamples +
                        chorus->counter - chorus->lookup_tab[i][chorus->phase[i]]) %
                        chorus->maxsamples] * chorus->decay[i];
                /* Adjust the output volume and size to 24 bit */
                d_out = d_out * chorus->out_gain;
                out = SOX_24BIT_CLIP_COUNT((sox_sample_t) d_out, effp->clips);
                *obuf++ = out * 256;
                /* Mix decay of delay and input */
                chorus->chorusbuf[chorus->counter] = d_in;
                chorus->counter =
                        ( chorus->counter + 1 ) % chorus->maxsamples;
                for ( i = 0; i < chorus->num_chorus; i++ )
                        chorus->phase[i]  =
                                ( chorus->phase[i] + 1 ) % chorus->length[i];
        }
```

L'amélioration directe de l'algorithme paraissant compliqué au premier abord, j'ai dans un premier temps décidé de paralléliser des parties de cette fonction. Cela n'a pas fonctionné, j'ai ensuite décider de me pencher sur l'algorithme à nouveau.

L'optimisation finale est la suite de ce rapport, les tentatives sont disponibles en annexe.

Implémentation de l'optimisation fonctionnelle :

A plusieurs endroit du code, notamment dans la partie critique qui nous intéresse, il y a plusieurs endroit des calculs ayant la forme suivante :

```
variable1 = (variable1 + 1) % variable2
```

Si `variable2` est forcement plus grande que `variable1`, au début de ce calcul, on peut remplacer cette formule par la suivante :

```
variable1 = variable1 + 1
if variable1 = variable2
    variable1 = 0
```

Ce qui, instinctivement, me parait moins compliqué comme tâche. Cela se confirmera par la suite.

La partie du code nous intéressant est la suivante :

```
chorus->counter = ( chorus->counter + 1 ) % chorus->maxsamples;
for ( i = 0; i < chorus->num_chorus; i++ )
        chorus->phase[i]  = ( chorus->phase[i] + 1 ) % chorus->length[i];
```

Pour rappel, la dernière ligne représente à elle seule près de 20% du temps de calcul et 15% des caches misses.

Il nous faut à présent montrer que `chorus->counter` est toujours plus petit que `chorus->maxsamples` et que `chorus->phase[i]` est toujours plus petit que `chorus->length[i]`.

`chorus->counter` est initialisé à 0 dans `sox_chorus_start`. `chorus->maxsamples` est également initialisé dans `sox_chorus_start` mais contient le plus grand des `chorus->sample`, eux même dérivés d'une adition suivi d'un multiplication puis d'une division de nombres positifs, le résultat est donc forcément positif et plus grand que 0. On peut donc afirmer que `chorus->counter` est toujours plus petit que `chorus->maxsamples` au lancement de notre code.

Chaque éléments du tableau `chorus->phase` sont initialisés à 0 dans `sox_chorus_start`. Les éléments du tableau `chorus->length` sont aussi initialisé dans la même fonction et sont positifs car ils découlent d'une division entre deux nombres positifs, le résultat est donc forcément positif et plus grand que 0. On peut donc affirmer que `chorus->phase[i]` est toujours plus petit que `chorus->length[i]` au lancement de notre code.

Voici maintenant notre code optimisé :

```
//optimisation
chorus->counter += 1;
if(chorus->counter = chorus->maxsamples)
    chorus->counter = 0;
for ( i = 0; i < chorus->num_chorus; i++ ){
    chorus->phase[i] += 1;
    if(chorus->phase[i] = chorus->length[i])
        chorus->phase[i] = 0;
}
```

Résultat du Benchmark de l'optimisation

Comparons, pour commencer, la version originale et notre version optimisée avec l'aide de `time` :

Version originale :

```
$ time ./sox input/test.wav result/test.wav chorus 1.0 0.4 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s

real    0m1.239s
user    0m1.149s
sys        0m0.094s
```

Version optimisée :

```
$ time ./sox input/test.wav result/test.wav chorus 1.0 0.4 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s

real    0m0.687s
user    0m0.610s
sys        0m0.081s

$ time ./sox input/test.wav result/test.wav chorus 1.0 0.4 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s

real    0m0.701s
user    0m0.616s
sys        0m0.087s
```

On utilise ici 2 chorus (ce qui n'était pas le cas lors du benchmark initial) pour accentuer l'optimisation, car la zone optimisée est dans une boucle affectée par le nombre de chorus.

Voici le résultat avec 4 chorus :

originale :

```
$ time ./sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0 0.5 0.8 1.0 -s 22.0 0.9
0.5 1.0 -s

real    0m1.898s
user    0m1.794s
sys        0m0.108s
```

optimisée :

```
$ time ./sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0 0.5 0.8 1.0 -s 22.0 0.9
0.5 1.0 -s

real    0m0.842s
user    0m0.778s
sys        0m0.067s
```

On peut déjà y voir une nette amélioration. La version optimisée est plus de 2 fois plus rapide que la version originale.

Continuons maintenant avec `stat` de PERF :

originale :

```
$ sudo perf stat -e cache-misses,cpu-cycles,mem-loads,mem-stores ./sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.
0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s

 Performance counter stats for './sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0
 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s':

         3’686’547      cache-misses
     5’697’340’014      cpu-cycles
                0      mem-loads
       320’803’219      mem-stores

       1.917701160 seconds time elapsed

       1.812081000 seconds user
       0.106207000 seconds sys

$ sudo perf stat -e cache-misses,cpu-cycles,mem-loads,mem-stores ./sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.
0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s

 Performance counter stats for './sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0
 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s':

         3’904’702      cache-misses
     5’738’326’094      cpu-cycles
                0      mem-loads
       321’345’680      mem-stores

       1.899597383 seconds time elapsed

       1.818213000 seconds user
       0.083156000 seconds sys
```

optimisée :

```
$ sudo perf stat -e cache-misses,cpu-cycles,mem-loads,mem-stores ./sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.
0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s

 Performance counter stats for './sox input/test.wav result/test.wav chorus 1.0 0.2 30.0 0.6 1.0 1.0 -s 20.0 0.4 1.0 1.0 -s 50.0
 0.5 0.8 1.0 -s 22.0 0.9 0.5 1.0 -s':

         1’320’095      cache-misses
     2’956’519’617      cpu-cycles
                0      mem-loads
       320’076’336      mem-stores

       0.890078250 seconds time elapsed

       0.816966000 seconds user
       0.076490000 seconds sys
```

Comme on peut le voir, l'optimisation a bien fonctionné, cette fonction est maintenant plus de 2 fois plus rapide et 2 fois moins gourmande en ressources CPU. De plus, même s'il n'y a pas de différences au niveau des `mem-stores`, on a baissé les caches misses d'un facteur 3.

Conclusion

Ce projet était très difficile à prendre en main. Mes premières suppositions de partir sur le parallélisme n'était pas bonnes : Dans le but de gagner du temps, je n'ai pas prêté assez attention à l'algorithme et j'ai essayé des solutions qui ne pouvaient pas fonctionner. Cela m'a fait perdre beaucoup de temps.

Une fois fait machine arrière, comprendre l'algorithme s'est révélé très intéressant et l'optimisation s'est révélée d'elle même. J'aurais du partir de là dès le départ.

Néanmoins, je suis très content d'avoir effectué ce projet et d'avoir trouvé cette optimisation. Les chiffres obtenus me paraissent incroyable et j’attends avec impatience votre retour.

Concernant la PR, je ne l'ai pas effectuée. J'aimerai la faire, mais je préfère attendre votre retour avant de déranger la communauté pour les raisons suivantes :

- Les chiffres obtenus sont presque trop incroyables, je préfère attendre que vous me confirmiez que c'est juste.
- Je ne sais pas comment faire vu que le logiciel n'est pas sur github et les recherches me demanderai un temps que je n'ai pas étant donnée que j'ai encore, entre autre, le projet 2 à effectuer pour dimanche.

Annexe : implémentation non-fonctionnelles

Un premier essai fut de tenter de paralléliser le calcul du chœur en lui même.

```
static int sox_chorus_flow(sox_effect_t * effp, const sox_sample_t *ibuf, sox_sample_t *obuf,
                   size_t *isamp, size_t *osamp)
{
        priv_t * chorus = (priv_t *) effp->priv;
        int i;
        //float d_in, d_out;
        //sox_sample_t out;
        size_t len = min(*isamp, *osamp);
        *isamp = *osamp = len;

        //ajout optimisation
        float d_out_temp[NUM_THREADS][PAD], d_in_temp[NUM_THREADS][PAD];
    omp_set_num_threads(NUM_THREADS);
    int nthreads;
     #pragma omp parallel
        {
        int i, j, id, nthrds, counter;
        long phase[chorus->num_chorus];
        sox_sample_t out;
        id = omp_get_num_threads();
        counter = chorus->counter+id;
        for ( i = 0; i < chorus->num_chorus; i++ )
            phase[i] = chorus->phase[i];
        nthrds = omp_get_num_threads();
        if (id==0) nthreads = nthrds;
        for (j = id; j<len; j=j+nthrds) {
                /* Store delays as 24-bit signed longs */
                d_in_temp[id][0] = (float) *(ibuf+j) / 256;
                /* Compute output first */
                d_out_temp[id][0] = d_in_temp[id][0] * chorus->in_gain;

                for ( i = 0; i < chorus->num_chorus; i++ )
                        d_out_temp[id][0] += chorus->chorusbuf[(chorus->maxsamples +
                        counter - chorus->lookup_tab[i][chorus->phase[i]]) %
                        chorus->maxsamples] * chorus->decay[i];
                //for(i=0; i<nthreads; i++) d_out+=d_out_temp[i][0];
                /* Adjust the output volume and size to 24 bit */
                d_out_temp[id][0] = d_out_temp[id][0] * chorus->out_gain;
                out = SOX_24BIT_CLIP_COUNT((sox_sample_t) d_out_temp[id][0], effp->clips);
                *(obuf+j) = out * 256;
                /* Mix decay of delay and input */
                chorus->chorusbuf[counter] = d_in_temp[id][0];
                counter =
                        ( counter + nthrds ) % chorus->maxsamples;
                for ( i = 0; i < chorus->num_chorus; i++ )
                    //#pragma omp critical
                        phase[i]  =( phase[i] + nthrds ) % chorus->length[i];
        }
        }
        /* processed all samples */
        return (SOX_SUCCESS);
}
```

Le problème de cette technique est que l'algorithme utilise les données précédentes pour calculer la prochaine valeurs. Il n'est donc pas parallélisable. Le programme si dessus fournis en sortie un fichier son qui n'est pas correct. Qui plus es, il n'augmente pas du tout la vitesse d'exécution et n'est donc pas plus optimale. Cela est surement du au fait que le compilateur optimise déjà le programme en le parallélisant à la compilation.

Nous allons essayer à présent de paralléliser une partie du programme qui peut l'être.

La fonction `sox_chorus_flow` contient le code suivant :

```
for ( i = 0; i < chorus->num_chorus; i++ )
                        chorus->phase[i]  =
                                ( chorus->phase[i] + 1 ) % chorus->length[i];
```

Cette partie là semble plus parallélisable et nous allons tenter de la faire avec des instructions SIMD.

Une première solution serait la suivante :

```
//optimisation : phase = long = 32 bits
                int i, nb;
                __m128i v1, v2, vsum;
                vsum = _mm_setzero_si128();
                v2 = _mm_set1_epi32(1);
                nb = (chorus->num_chorus) - (chorus->num_chorus % 4);
                for ( i = 0; i < chorus->num_chorus; i+=4 ){
                    v1 = _mm_loadu_si128(&chorus->phase[i]);
                    v1 = _mm_add_epi32(v1, v2);
                    _mm_storeu_si128(&chorus->phase[i], v1);
                }
                for (i = nb; i < chorus->num_chorus; i++){
                    chorus->phase[i]  = ( chorus->phase[i] + 1 );
                }
```

Le cas du modulo s'est révélé plus compliqué à faire avec des SMID que prévu, il a été décidé de le déplacer à l'extérieur de cette boucle pour éviter cette complication.

Malheureusement cela ne fonctionne pas. Le programme produit une segmentation fault et je n'ai pas réussi à trouver la solution à ce problème.

Commençant gentiment à tourner en rond avec le parallélisme, et ayant perdu beaucoup de temps à tenter de le faire fonctionner, je décide de me pencher sur l'algorithme car à l'heure ou j'écris ces lignes la deadline du projet est déjà dépassée.