

Introduction

Présentation du sujet

Nous avons travaillé sur le sujet 13.

Le projet est séparé en 2 phases majeures :

- une phase d'optimisation statique ; nous ne modifions pas le code et cherchons à l'optimiser lors de la compilation
- une phase d'optimisation dynamique ; ici nous optimisons le code avant de le compiler

Nous cherchons à obtenir les meilleures performances possibles en modifiant différents paramètres.

Pour ce faire nous allons donc jouer sur les paramètres suivants :

- compilation sous deux compilateurs : gcc et icx
- choix des flags judicieux en fonction de notre code
- choix des warmups et répétitions
- choix de la taille des tableaux en fonction de différents caches et RAM.

Fonction à optimiser :

```
void s13 (unsigned n , const float a[n] ,const float b[n] , float c[n][n]
,int offset , double radius) {
    int i, j;

    for ( i = 0; i < n ; i ++) {
        for ( j = 0; j < n ; j ++) {
            if ( offset + j < 0 || offset + j >= n )
                continue;

            c [ i ][ offset + j ] = 0.0;

            if ( a[offset + j] < radius ) {
                c [ i ][ offset + j ] = a [ offset + j ] / b [ i ];
            }
        }
    }
}
```

I Méthodologie

I.1) Justification des 5 points clés du driver (warmups, repets etc.)

Le driver tente de résoudre des problèmes de stabilité du système qui pourraient fausser les mesures de performance du kernel.

Il prend donc différentes mesures pour résoudre ces problèmes :

Répétitions de "warmup"

Ce sont des répétitions qui viennent en amont des mesures afin de préparer le système.

Ces warmups sont des passages "à vide" du code que l'on veut benchmarker.

Cela permet de remplir les caches avec les valeurs des tableaux de données sur lesquelles le kernel fait ses opérations.

Cela permet de "chauffer" la machine pour effectuer les tests dans des conditions les plus stables possibles.

On verra l'utilité des warmups dans les figures suivantes, qui montrent que la première itération du kernel s'effectue dans un laps de temps beaucoup plus grand que les suivantes.

Elles permettent d'exclure le régime transitoire. Elles sont d'autant plus importantes lorsque la machine est froide.

Lors des mesures, ces répétitions amortissent l'erreur du timer.

Répétitions de l'expérience

Cette mesure prise par le driver pour réduire les erreurs de mesures est simple à comprendre : on effectue plusieurs fois les mesures pour en faire la médiane et atténuer les éventuelles perturbations qui pourraient fausser les mesures.

Méta-répétitions pour mesurer la stabilité

Comme le point ci-dessus, cette répétition va exécuter N fois tout le processus de test : les warmups + les répétitions

Cela permet encore une fois d'obtenir

- Répétitions du corps du driver
- Via un script ou une boucle dans le driver
- En toute rigueur, 31 méta-répétitions nécessaires
- Souhaité : $(\text{médiane} - \text{minimum}) / \text{minimum} < 5\%$ => on prendre NB_META 31

Sonde utilisée

On utilisera la fonction rdtac() de unistd.h pour les mesures de temps.

Valeurs des tableaux

Les valeurs du tableau sont randomisées à chaque meta-répétition.

La machine prend donc le temps de re-remplir ses caches avec les nouvelles valeurs grâce au

I.2) Stratégies utilisées en commun pour améliorer la stabilité

Optimisation de la compilation

`gcc` propose plusieurs "flags" de compilation.

On se propose dans un premier temps de trouver une bonne combinaison de flags pour améliorer la performance de notre kernel.

On trouve les options disponibles suivantes intéressantes :

```
# basic
-O1
-O2
-O3
-O3 -ffast-math -fstack-arrays
-march=native

# floating point computing (can be implmed by -march=native)
-msse4.2 -mavx / -msse2avx # /proc/cpuinfos | grep sse
-mfpmath=sse
-mmx

-fallow-store-data-races
```

Pour vérifier leur performance effective, on compile une version différente de l'exécutable, chacune avec des flags différents.

On exécute tous les executables :

```
CORE_ID=3 # the core id on which the bench is executed
cpupower -c $CORE_ID frequency-set --governor performance

size=1024
warmup=100
rep=100

for exe in `find . -maxdepth 1 -executable -type f ! -name "*..*"; do
    taskset -c $CORE_ID $exe $size $warmup $rep >
    exec_output/${exe}_${iteration}.dat
done
```

On compare maintenant facilement les résultats entre:

```
for file in `ls exec_output/`; do
    echo `awk '{ sum += $1 } END { print sum }' $file` $file
done | sort
```

Ce qui nous sort :

```
65.35 s13_03n_v5.dat # -O3 -msse4.2 -mavx -march=native
65.56 s13_03n_v4.dat # -O3 -mfpmath=sse -march=native
66.64 s13_03n.dat # -O3 -march=native
67.86 s13_03n_v2.dat # -O3 -ffast-math -fstack-arrays -march=native
68.3 s13_03n_v3.dat # -O3 -march=native -mmmx
73.12 s13_03.dat # -O3
73.85 s13_02.dat # -O2
74.53 s13_01.dat # -O1
```

On remarque que les flags `-mfpmath=sse -msse4.2 -mavx` et `-march=native` sont particulièrement efficaces.

On gardera ces flags pour la compilation avec `gcc`.

Intel OneAPI

Nos machines ayant des processeurs intel, il est intéressant d'utiliser le compilateur d'Intel : `icx`. Oneapi est censé produire un binaire plus optimisé, en proposant une compilation plus agressive, et optimisé pour les processeurs intel.

Optimisation code

Optimisations évidentes

Dans un premier temps, on optimise le kernel juste en comprenant le code.

On trouve une forme plus "agréable" en supprimant des conditions inutiles :

```
void s13 (unsigned n, const float a[n], const float b[n], float c[n][n], int
offset, double radius) {
    int i, j;

    for ( i = 0; i < n ; i ++ ) {
        for ( j = offset; j < n; j ++ ) {

            if ( a[ j ] < radius ) {
                c [ i ][ j ] = a [ j ] / b [ i ];
            }
            else {
                c [ i ][ j ] = 0.0;
            }

        }
    }
}
```

On vérifie que notre version a le même comportement que l'originale en compilant les versions originales et corrigées, et en ajoutant une fonction `dump_result()` et en comparant les sorties avec `diff`.

On compare maintenant les performances de la nouvelle implémentation avec leur temps d'exécution des deux versions

Avec la même méthode que pour l'optimisation des flags de gcc on trouve :

```
52.12 CORRECTED_s13_03n_v6.dat
54.31 CORRECTED_s13_03n_v5.dat
55.34 CORRECTED_s13_03n_v4.dat
61.93 ORIGINAL_s13_03n_v5.dat
63.94 ORIGINAL_s13_03n_v6.dat
64.09 ORIGINAL_s13_03n_v4.dat
```

On remarque que la version corrigée de la fonction est de loin meilleure à la version originale. Au passage, on vérifie que les flags de gcc sont bien intéressants niveau performance.

I.3) Méthodologie de détermination du nb de répétitions de warmup et demesure

On cherche

$(\text{médiane} - \text{minimum}) / \text{minimum} < 5 \% = 31$

II Analyse statique MAQAO"

Intéressons nous à l'assembleur produit par les deux compilateurs : `gcc` et `icc`.

Intel OneAPI `icc`

Tous les binaires produits ne sont pas vectorisés. Maqao indique en effet que le taux de vectorisation pour la boucle principale varie entre 0 et 15% pour les meilleurs flags de compilation.

Ainsi, maqao indique qu'on pourrait avoir un speedup en moyenne de 7.77% en vectorisant complètement les instructions pour tous les binaires.

Un gros enjeu de la seconde partie du projet sera donc de réussir à modifier le code de la boucle de manière à ce que le code soit valorisable, et que le compilateur puisse le comprendre rapidement pour mettre en place des instructions vectorielles.

Une analyse des binaires avec `radare2` nous confirme qu'aucune vectorisation n'est effectuée :

- les instructions utilisées sont de type scalaire (eg. `addss`, `movss`, ...).
- on utilise les registres `xmm`



[0x00401580]> 0x401580 # sym.s13 (signed int64_t arg1, void *arg2, void *arg4, int64_t

```
0x4015d1 [oh]
; moves data from src to dst
eax = ebp
; moves data from src to dst
dword [r14 + rax*4] = 0
; arg2
; move scalar single-fp values
movss xmm1,dword [rsi + rax*4]
; bitwise logical xor for single-fp values
xmm2 ^= xmm2
; convert scalar single-fp value to scalar double-fp value
cvtss2sd xmm2,xmm1
; unordered compare scalar double-fp values and set eflags
ucomisd xmm0,xmm2
; jump short if below or equal/not above (cf=1 or zf=1)
if (((unsigned) var) <= 0) goto 0x4015c0
```

```
0x4015ed [oi]
; divide scalar single-fp values
divss xmm1,dword [rdx + r11*4]
; move scalar single-fp values
movss dword [r14 + rax*4],xmm1
; jump
goto 0x4015c0
```

```
0x4015c0 [oe]
; CODE XREFS from sym.s13 @ 0x4015cb, 0x4015cf, 0x4015eb, 0x4015f9
; adds src and dst, stores result on dst
ebp += 1
; adds src and dst, stores result on dst
rbx += 0xffffffffffffffff
; jump short if equal (zf=1)
if (!var) goto 0x4015a0
```

```
[0x00401540]> 0x401540 # sym.s13 (uint32_t arg1, int64_t arg2, int64_t arg3,
; jump short if above or equal (cf=0)
jae 0x401640
```

f t

```
0x401653 [ok]
; moves data from src to dst
ebx = ecx
; moves data from src to dst
dword [r13 + rbx*4] = 0
; arg2
; move scalar single-fp values
movss xmm1,dword [rsi + rbx*4]
; bitwise logical xor for single-fp values
xmm2 ^= xmm2
; convert scalar single-fp value to scalar double-fp value
cvtss2sd xmm2,xmm1
; unordered compare scalar double-fp values and set eflags
ucomisd xmm0,xmm2
; jump short if below or equal/not above (cf=1 or zf=1)
if (((unsigned) var) <= 0) goto 0x401640
```

f t

```
0x401670 [ol]
; divide scalar single-fp values
divss xmm1,dword [r12]
; move scalar single-fp values
movss dword [r13 + rbx*4],xmm1
; jump
goto 0x401640
```

v

Filters									
<input type="checkbox"/> Level	<input checked="" type="checkbox"/> Coverage (%)	<input type="checkbox"/> Time (s)	<input checked="" type="checkbox"/> Vectorization Ratio (%)	<input checked="" type="checkbox"/> Vectorization Efficiency (%)	<input checked="" type="checkbox"/> Speedup If No Scalar Integer	<input checked="" type="checkbox"/> Speedup If FP Vectorized	<input checked="" type="checkbox"/> Speedup If Fully Vectorized	<input type="checkbox"/> Speedup If Perfect Load Balancing	
<input type="checkbox"/> Stride 0	<input type="checkbox"/> Stride 1	<input type="checkbox"/> Stride n	<input type="checkbox"/> Stride Unknown	<input type="checkbox"/> Stride Indirect	Select none				
Loop id	Source Location	Source Function	Coverage (%)	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	
11	s13_icx_ultimate -	s13	96.81	11.43	18.57	1.21	1.69	5.25	
10	s13_icx_ultimate -	s13	1.06	5.05	20.34	1.26	1.17	6.13	

gcc

Les résultats de l'analyse des binaires de gcc avec Maqao sont très similaires avec ceux précédents pour `icc`.

Les codes ne sont toujours pas vectorisés.

[0x00001950]> 0x1950 # sym.s13 (size_t arg1, size_t arg2, size_t arg3, size_t arg4, si

f t

```
0x1990 [of]
; arg4
; moves data from src to dst
dword [rcx + rdx] = 0
; move scalar single-fp values
movss xmm0,dword [r9 + rdx]
; logical exclusive or
xmm1 ^= xmm1
; convert scalar single-fp value to scalar double-fp value
cvtss2sd xmm1,xmm0
; compare scalar ordered double-fp values and set eflags
comisd xmm2,xmm1
; jump short if below or equal/not above (cf=1 or zf=1)
if (((unsigned) var) <= 0) goto 0x19b5
```

f t

```
0x19ab [og]
; divide scalar single-fp values
divss xmm0,dword [r8]
; arg4
; move scalar single-fp values
movss dword [rcx + rdx],xmm0
```

v

```
0x19b5 [oh]
; CODE XREFS from sym.s13 @ 0x198a, 0x198e, 0x19a9
; adds src and dst, stores result on dst
eax += 1
; adds src and dst, stores result on dst
rdx += 4
; compare two operands
var = eax - esi
; jump short if not equal/not zero (zf=0)
if (var) goto 0x1988
```

Filters										?
<input checked="" type="checkbox"/> Level	<input checked="" type="checkbox"/> Coverage (%)	<input type="checkbox"/> Time (s)		<input checked="" type="checkbox"/> Vectorization Ratio (%)	<input checked="" type="checkbox"/> Vectorization Efficiency (%)	<input checked="" type="checkbox"/> Speedup If No Scalar Integer	<input checked="" type="checkbox"/> Speedup If FP Vectorized	<input checked="" type="checkbox"/> Speedup If Fully Vectorized	<input type="checkbox"/> Speedup If Perfect Load Balancing	
<input type="checkbox"/> Stride 0	<input type="checkbox"/> Stride 1	<input type="checkbox"/> Stride n	<input type="checkbox"/> Stride Unknown	<input type="checkbox"/> Stride Indirect	Select all					
Loop id	Source Location	Source Function	Level	Coverage (%)	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	
8	s13_03n_v6	s13	Innermost	99.57	0	16.02	1.18	1.18	6.15	

On remarque cependant qu'en utilisant des flags d'optimisation plus agressifs, `icx` divise la boucle principale en deux contrairement a `gcc` qui le garde en une seule boucle.

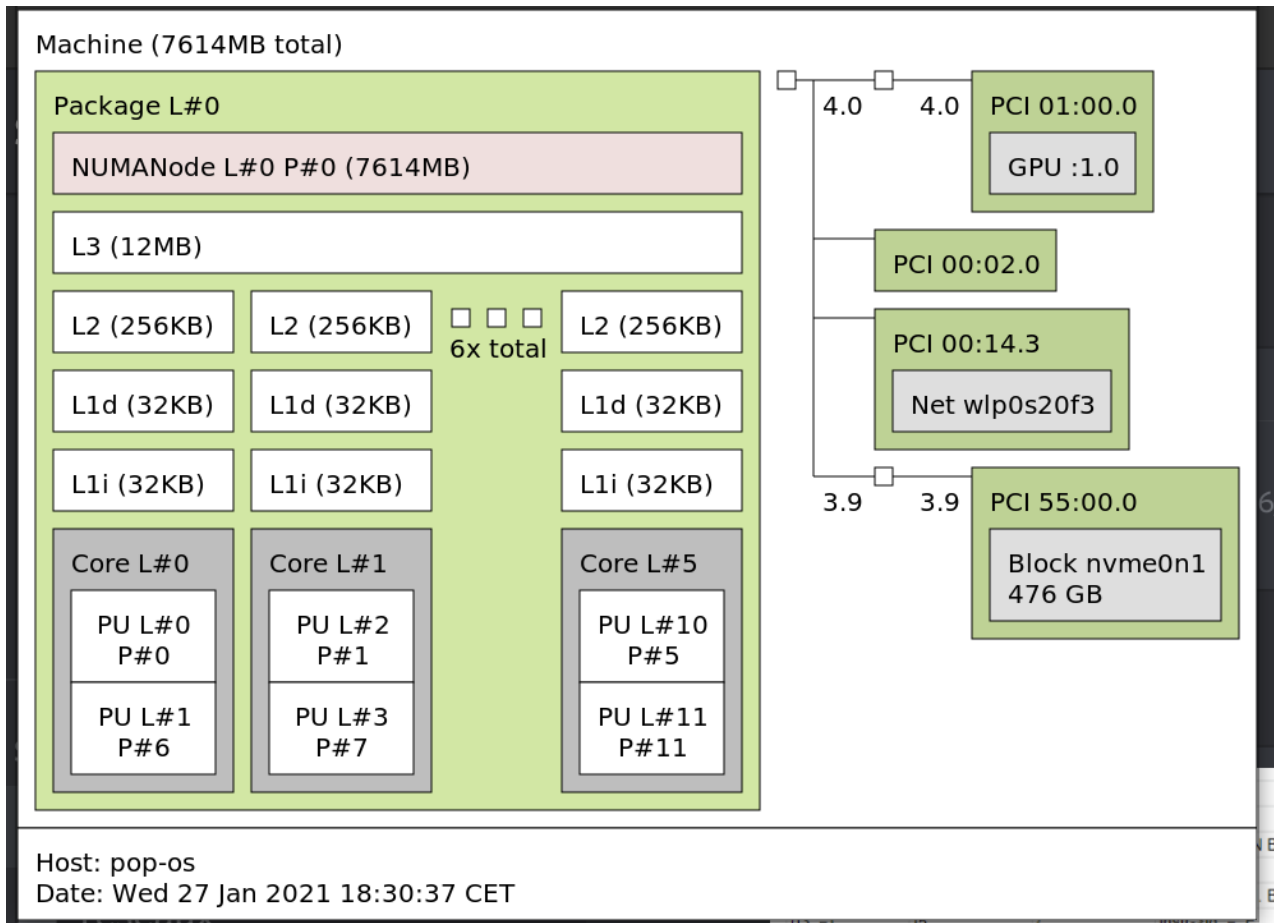
III Cache L1

III.1) Environnement expérimental

Toutes les mesures ont été effectuées sur la machine suivante :

Type	Informations
processeur	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Génération du processeur	Comet Lake
Linux	Pop!_OS 20.10 x86_64
Virtualisation	Simple-boot
Version noyau	Linux version 5.8.0-7642-generic
Version GCC	gcc 10.2.0
Version oneAPI	Intel(R) oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113)
Version MAQAO	2.13.0
Techno RAM	2x8 Go DDR4 3200MHz

Voici les informations des différents caches de la machine.



III.2) Justification de la taille des tableaux pour que ça tienne dans le cache

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité.

Nous traitons 3 tableaux : 2 de taille n et 1 de taille n^2 .

De ce fait il faut que $2n + n^2 < \text{cache cible}$.

Choix de n en fonction de L1

Notre cache L1 fait 32 kB. Nous allons donc faire en sorte de le remplir à 70%.

On note $L1 = \text{sizeof}(L1) * 70\% / \text{sizeof}(\text{float})$.

On cherche donc n tel que $2n + n^2 = L1$.

On trouve donc $n = \text{sqrt}(5601) - 1$

$$n = 73$$

III.3) Détermination du nombre de répétitions de warmups et de mesures

Nous utilisons la fonction RDTSC pour avoir un outil qui nous permet de déduire le nombre de cycles pour chaque répétition.

Pour déterminer le nombre de warmup nous avons utilisé la fonction clock() afin de connaître le temps d'exécution des warmups. Nous avons changé le nombre de warmup jusqu'à obtenir un temps d'exécution des warmups d'environ 1 seconde, afin d'obtenir le nombre optimal de warmup.

```
sudo cpupower -c 3 frequency-set --governor performance;taskset -c 3 ./s13_03n_v6 73 150000 50
```

```
Setting cpu: 3
temps : 0.958867
```

Ci-dessus seul argv[1] et argv[2] nous intéressent car argv[1] correspond au cache utilisé (ici L1), puis argv[2] au nombre de warmup pour pouvoir atteindre un temps d'exécution du warmup d'environ 1 seconde comme attendu. Ici argv[3] nous est inutile dans la détermination du nombre warmup.


III.4) Mesures des diverses variantes et analyse

On observe au niveau du cache L1 pour gcc qu'avec le flags O1 le temps total d'exécution est 3.74 secondes alors qu'avec les flags O2, O3 et O3n (O3 + march=native) le temps total est d'environ 1 seconde. On remarque aussi que l'Array Access Efficiency (%) est à 83% pour O1 et 95% pour les flags O2,O3 et O3n. On peut donc en conclure qu'avec le compilateur gcc les flags O2, O3 et O3n permettent d'améliorer l'efficacité de l'exécution.

Avec le compilateur icc on observe que le temps total d'exécution est d'environ 0.9 secondes et que l'Array access Efficiency est de 100%, peu importe les flags utilisés. On peut donc en conclure que pour le cache L1 le compilateur ICC est plus performant que le compilateur GCC.

s13_01 - 2021-03-16 22:40:06 - MAQAO 2.13.0

Help is available by moving the cursor above any ? symbol or by checking [MAQAO website](#).

Global Metrics		?	CQA Potential Speedups Summary	
Total Time (s)	3.74			
Profiled Time (s)	3.74			
Time in loops (%)	100			
Time in innermost loops (%)	99.06			
Time in user code (%)	100			
Compilation Options	Not Available			
Perfect Flow Complexity	7.30			
Array Access Efficiency (%)	83.33			

Analyse maqao cache L1 GCC flag O1

Help is available by moving the cursor above any ? symbol or by checking [MAQAO website](#).

Global Metrics		CQA Potential Speedups Summary
Total Time (s)	1.06	
Profiled Time (s)	1.06	
Time in loops (%)	100	
Time in innermost loops (%)	98.6	
Time in user code (%)	100	
Compilation Options	Not Available	
Perfect Flow Complexity	5.74	
Array Access Efficiency (%)	95.83	

Analyse maqao cache L1 GCC flag O3n

Help is available by moving the cursor above any ? symbol or by checking [MAQAO website](#).

Global Metrics		CQA Potential Speedups Summary
Total Time (s)	0.89	
Profiled Time (s)	0.89	
Time in loops (%)	100	
Time in innermost loops (%)	98.91	
Time in user code (%)	100	
Compilation Options	Not Available	
Perfect Flow Complexity	5.80	
Array Access Efficiency (%)	100.00	

Analyse maqao cache L1 ICC flag mtune

IV Cache L2

IV.1) Justification de la taille des tableaux pour que ça tienne dans le cache L2

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité.

Nous traitons 3 tableaux, 2 de taille n et 1 de taille n^2 .

De ce fait il faut que $2n + n^2 < \text{cache choisi}$.

Notre cache L2 fait 256 kB. Nous allons donc faire en sorte de le remplir à 70%.

On note $L2 = (L1 + \text{sizeof}(L2) * 70\%) / \text{sizeof}(\text{float})$.

On cherche donc n tel que $2n + n^2 = L2$.

On trouve donc $n = \sqrt{52801} - 1$

$$n = 228$$

IV.2) Détermination du nombre de répétitions de warmups et de mesures

Pour déterminer le nombre de warmup nous avons utilisé la fonction `clock()` afin de connaître le temps d'exécution des warmups. Nous avons changé le nombre de warmup jusqu'à obtenir un temps d'exécution des warmups d'environ 1 seconde, afin d'obtenir le nombre optimal de warmup.

```
sudo cpupower -c 3 frequency-set --governor performance;taskset -c 3 ./s13_03n_v6 228
20000 50
```

```
Setting cpu: 3
temps : 1.005044
```

Ci-dessus seul argv[1] et argv[2] nous intéressent car argv[1] correspond au cache utilisé (ici L2), puis argv[2] au nombre de warmup pour pouvoir atteindre un temps d'exécution du warmup d'environ 1 seconde comme attendu. Ici argv[3] nous est inutile dans la détermination du nombre warmup.

IV.3) Mesures des diverses variantes et analyse

On observe au niveau du cache L2 pour gcc qu'avec le flags O1 le temps total d'exécution est 6.2 secondes alors qu'avec les flags O2 le temps est de 5 secondes, O3 et O3n (O3 + march=native) le temps total est d'environ 2 seconde. On peut donc en conclure qu'avec le compilateur gcc les flags O3 et O3n permettent d'améliorer l'efficacité de l'exécution, et que le flag O2 est quand même plus efficace que le flag O1.

V Cache L3

V.1) Justification de la taille des tableaux pour que ça tienne dans le cache L3

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité.

Nous traitons 3 tableaux, 2 de taille n et 1 de taille n².

De ce fait il faut que $2n + n^2 < \text{cache choisi}$.

Notre cache L3 fait 12 MB. Nous allons donc faire en sorte de le remplir à 70%.

On note $L3 = (L1 + L2 + \text{sizeof}(L3) * 70\%) / \text{sizeof}(\text{float})$.

On cherche donc n tel que $2n + n^2 = L3$.

On trouve donc $n = \sqrt{282001} - 1$

$n = 530$

V.2) Détermination du nombre de répétitions de warmups et de mesures

Pour déterminer le nombre de warmup nous avons utilisé la fonction clock() afin de connaître le temps d'exécution des warmups. Nous avons changé le nombre de warmup jusqu'à obtenir un temps d'exécution des warmups d'environ 1 seconde, afin d'obtenir le nombre optimal de warmup.

```
sudo cpupower -c 3 frequency-set --governor performance; taskset -c 3 ./s13_03n_v6 530
3000 50
```

```
Setting cpu: 3
temps : 0.971788
```

Ci-dessus seul argv[1] et argv[2] nous intéressent car argv[1] correspond au cache utilisé (ici L3), puis argv[2] au nombre de warmup pour pouvoir atteindre un temps d'exécution du warmup d'environ 1 seconde comme attendu. Ici argv[3] nous est inutile dans la détermination du nombre warmup.

V.3) Mesures des diverses variantes et analyse

On remarque que comme pour les niveaux de caches précédent que les flags de GCC O3 et O3n sont plus performant que les flags O1 et O2.

VI RAM

VI.1) Justification de la taille des tableaux pour que ça tienne dans la RAM

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité.

Nous traitons 3 tableaux, 2 de taille n et 1 de taille n².

De ce fait il faut que $2n + n^2 < \text{cache choisi}$.

Nous voulons avoir un tableau passant dans la RAM. De ce fait nous allons remplir les 3 caches et faire en sorte de les saturer pour entrer dans la RAM.

On note $\text{RAM} = (L1 + L2 + 2*L3) / \text{sizeof(float)}$.

On cherche donc n tel que $2n + n^2 = \text{RAM}$.

On trouve donc $n = \sqrt{672001} - 1$

$n = 818$

VI.2) Détermination du nombre de répétitions de warmups et de mesures

Pour déterminer le nombre de warmup nous avons utilisé la fonction clock() afin de connaître le temps d'exécution des warmups. Nous avons changé le nombre de warmup jusqu'à obtenir un temps d'exécution des warmups d'environ 1 seconde, afin d'obtenir le nombre optimal de warmup.

```
sudo cpupower -c 3 frequency-set --governor performance;taskset -c 3 ./s13_03n_v6 818 1600 50
```

```
Setting cpu: 3
temps : 1.062283
```

Ci-dessus seul argv[1] et argv[2] nous intéressent car argv[1] correspond au cache utilisé (ici la RAM), puis argv[2] au nombre de warmup pour pouvoir atteindre un temps d'exécution du warmup d'environ 1 seconde comme attendu. Ici argv[3] nous est inutile dans la détermination du nombre warmup.

VI.3) Mesures des diverses variantes et analyse

On remarque que comme pour les niveaux de caches précédent que les flags de GCC O3 et O3n sont plus performant que les flags O1 et O2.

Conclusion

Malgré les différents flags et les différents compilateurs utilisés, nous avons pu remarquer à travers les différents rapports maqao que le code ne produit pas un binaire vectorisé. Nous observons donc la limite de l'étude statique de ce code, car le code original de notre fonction à optimiser ne permet pas aux compilateurs de l'induire vers une vectorisation des instructions.

Le fait d'utiliser des flags plus optimisés ne rend pas forcément le code plus performant. Les codes générés sont parfois identiques.

Par contre la gestion des niveaux de caches que nous avons calculés au préalable permet d'observer que le nombre de cycles par répétitions est plus important dans le cache L1 que dans la Ram (donc plus le n est petit plus le nombre de cycles par répétition est important).

Lors de la phase 2, il sera donc important d'optimiser le code avant la compilation afin de permettre de meilleurs résultats et analyses, typiquement le résultat attendu sera l'utilisation d'instructions vectorielles. Le fait d'optimiser le code au préalable permet des gains d'exécution importants.

Notre étude s'est focalisée dans un premier temps sur gcc. Les comparaisons faites entre icx et gcc peuvent donc ne pas être représentatives de la réalité.

C'est un axe majeur à améliorer lors de la phase 2.