Projet AOA sujet 10 ◎

ALEXANDRE Julien
julien.alexandre@isty.uvsq.fr

VIRLOGEUX Marin
marin.virlogeux@isty.uvsq.fr

LEDOYEN Paul
paul.ledoyen@isty.uvsq.fr

DRISSI Mohamed Reda
reda-mohamed@isty.uvsq.fr

2 avril 2018

Table des matières

I Introduction . . . 2
I.1 Objectifs . . . 2
I.2 Spécifications de la machine utilisée . . . 2
I.3 Système . . . 2
I.4 Topologie du système . . . 2
I.5 Noyau . . . 3
II Détermination des paramètres . . . 4
II.1 Taille des données . . . 4
II.2 Nombre de warmup . . . 4
II.2.1 L1 : taille 60 . . . 4
II.2.1.1 compilateur GCC© . . . 5
Optimisation O2 . . . 5
Optimisation O3 . . . 6
Optimisation O3 march=native . . . 7
Optimisation Ofast . . . 8
II.2.1.2 Compilateur ICC© . . . 9
Optimisation O2 . . . 9
Optimisation O3 . . . 10
Optimisation O3 xHost . . . 11
Optimisation Ofast . . . 12
II.2.2 L2 : taille 170, L3 : taille 1000, RAM : taille 20000 12
III Résultats et interprétations . . . 13
III.1 Compilateur GCC© . . . 13
III.1.1 Option O2 . . . 13
III.1.1.1 Assembleur . . . 13
III.1.1.2 MAQAO . . . 13
III.1.2 Option O3 . . . 13
III.1.2.1 Assembleur . . . 13
III.1.2.2 MAQAO . . . 14
III.1.3 Option O3Nat . . . 14
III.1.3.1 Assembleur . . . 14
III.1.3.2 MAQAO . . . 14
III.1.4 Option Ofast . . . 15
III.1.4.1 Assembleur . . . 15
III.1.4.2 MAQAO . . . 15

III.1.4.3 Justification 15
`Ofast` 15
`funroll-loops` 15
III.1.4.4 Optimisation `funroll-loops` 15
Assembleur 15
MAQAO 16
III.2 Compilateur ICC 17
III.2.1 `Option O2` 17
III.2.1.1 Assembleur 17
III.2.1.2 MAQAO 17
III.2.2 `Option O3` 18
III.2.2.1 Assembleur 18
III.2.2.2 MAQAO 18
III.2.3 `Option O3Nat` 18
III.2.3.1 Assembleur 18
III.2.3.2 MAQAO 19
III.2.4 `Option Ofast` 19
III.2.4.1 Assembleur 19
III.2.4.2 MAQAO 19
III.2.5 `Option funroll-loops` 19
III.2.5.1 Assembleur 20
III.2.5.2 MAQAO 20

I Introduction

I.1 Objectifs

L'objectif de ce projet est d'étudier un noyau de code et d'optimiser sa compilation. Nous traitons ici la phase I de ce projet qui consiste en

- mesurer la performance du noyau pour différents niveaux d'optimisations de GCC©et d'ICC
- trouver d'autres options d'optimisations pertinentes
- expliquer les différences de performances entre versions d'optimisation avec MAQAO ou likwid
- justifier l'implémentation du driver

I.2 Spécifications de la machine utilisée

Nous avons traité les différents cas sur la même machine pour conserver une certaine constance dans les mesures effectuées.

- CPU : intel core i7 6700K 4.0GHZ 4 physical cores, 8 logical(HyperThreading©) turbo boost off
- RAM : Corsair CMK16GX4M2B3000C15 Vengeance LPX 16GB DDR4 3000MHz C15 XMP 2.0
- Stockage : Samsung 850 PRO SSD 512GB

I.3 Système

- OS : Debian 9.4 Stretch (stable) x86_64
- Kernel : 4.9.0-6-amd64
- GCC© : 6.3.0 20170516 (Debian 6.3.0-18+deb9u1)
- ICC : 18.0.1 20171018

I.4 Topologie du système

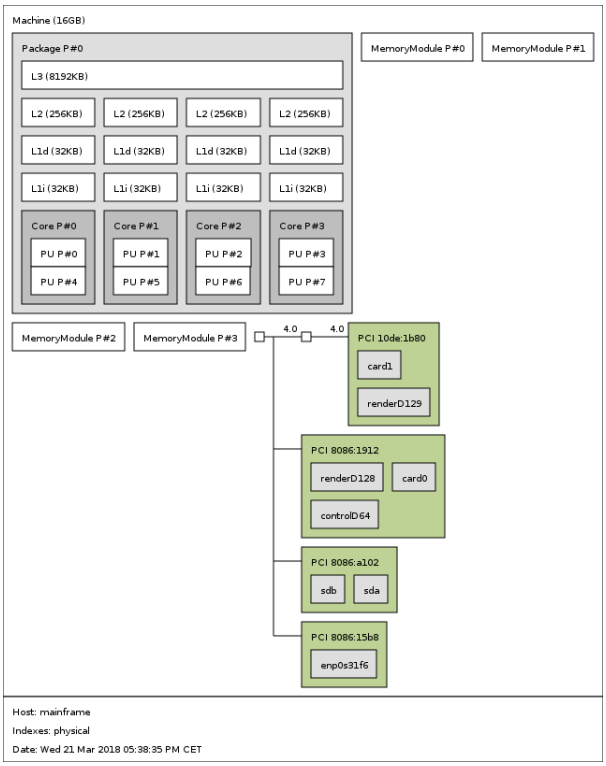

FIGURE 1 – Topologie générée par lstopo

I.5 Noyau

Le noyau traité est le suivant :

```
float baseline(int n,double a[n][n])
{
    int i,j;
    double s=0.0;
    for(j=0;j<n;j++)
        for(i=0;i<n;i++)
            s += a[j][i];
    return s;
}
```

II Détermination des paramètres

II.1 Taille des données

Notre boucle utilise un tableau de `double` de taille $n \times n$ chaque case prend 8 octets.
Donc le coût total (en mémoire) de notre boucle sera de $8n^2$.

Si nous voulons utiliser L1, L2, L3 ou la ram il faut trouver l'intervalle de chacun Soit T la taille maximale (qui serait en puissance de 2 alors $T = 2^t$) :

$$8n^2 \leq 2^t \tag{1}$$

$$n \leq 2^{\frac{t-3}{2}} \tag{2}$$

Les données des différents caches et ram sont
- L1 : 32Ko = 2^{15} octets
- L2 : 256Ko = 2^{18} octets
- L3 : 8192Ko = 2^{23} octets
- RAM : 16Gb = 2^{34} octets

Mémoire	2^t	Taille	coût
L1	15	64	31.64Ko
L2	18	181	256Ko
L3	23	1024	8190Ko
RAM	34	46340	16Gb

Par précaution, nous n'allons pas prendre des tailles de tableau exactement identiques aux tailles maximales des caches, pour éviter tout débordement.

II.2 Nombre de warmup

Nous allons étudier pour chaque taille de tableau, et pour chaque option d'optimisations le nombre de cycles nécessaires (ordonnée) pour chaque itération de warmup (abscisse). On pourra ainsi déterminer approximativement le nombre de warmup necessaires pour atteindre le régime permanent.

II.2.1 L1 : taille 60

II.2.1.1 compilateur `GCC`©

`Optimisation O2` Nous trouvons ce résultat pour L1 :

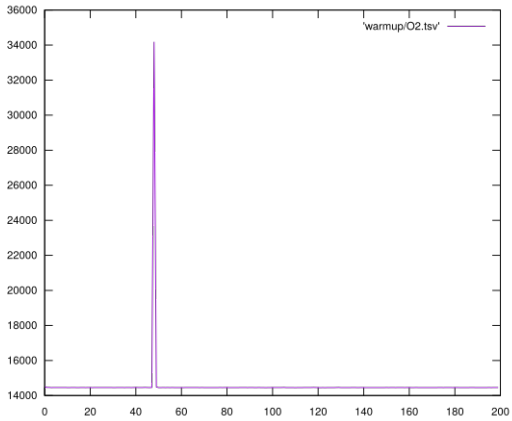

FIGURE 2 – Résultat du warmup avec option `O2` du compilateur **GCC**©

Le pic observé correspond à une interférence, nous n'en tiendrons donc pas compte. On observe que le régime permanent est atteint très vite, on appliquera donc 20 warmups.

`Optimisation O3` Nous trouvons ce résultat pour L1 :

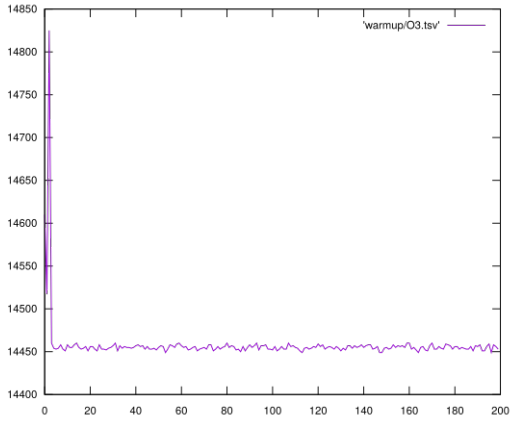

FIGURE 3 – Résultat du warmup avec option `O3` du compilateur **GCC**©

Ici on observe un régime transitoire de très faible durée, un nombre de warmups de 20 est également suffisant.

`Optimisation O3 march=native` Nous trouvons ce résultat pour L1 :

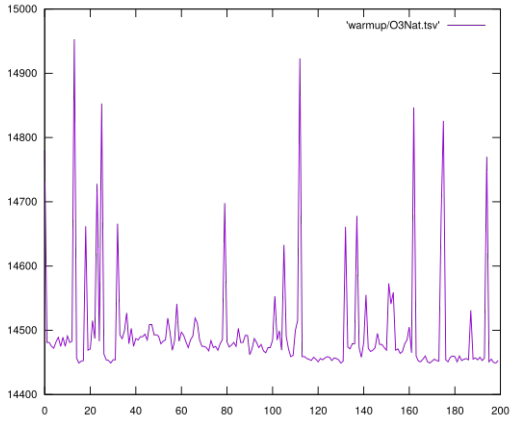

FIGURE 4 – Résultat du warmup avec option `O3 march=native` du compilateur **GCC©**

Le résultat obtenu ici est totalement instable, on ne peut pas identifier de stabilisation, donc opérer des warmups ne semble pas avoir de grand intérêt.

`Optimisation Ofast` Nous trouvons ce résultat pour L1 :

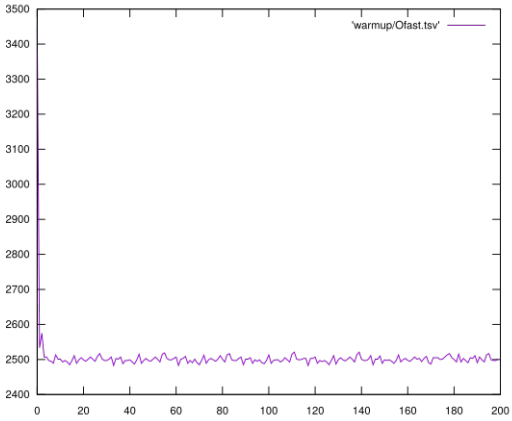

FIGURE 5 – Résultat du warmup avec option `Ofast` du compilateur **GCC©**

On observe, comme pour O3, un très court régime transitoire (inférieur à 5 cycles). Un premier warmup de 20 itérations est retenu.

II.2.1.2 Compilateur `ICC`©

`Optimisation O2` Nous trouvons ce résultat pour L1 :

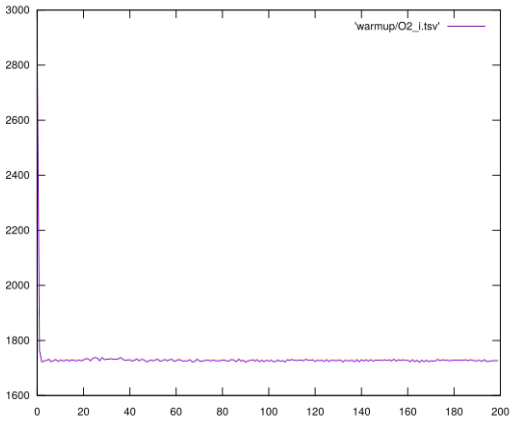

FIGURE 6 – Résultat du warmup avec option `O2` du compilateur **ICC**
Très court régime transitoire, on conserve le nombre de 20 warmups.

`Optimisation O3` Nous trouvons ce résultat pour L1 :

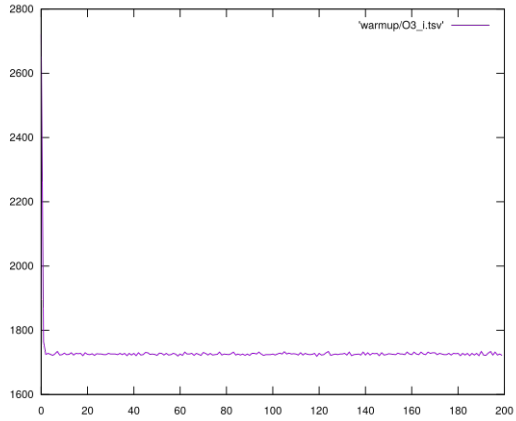

FIGURE 7 – Résultat du warmup avec option `O3` du compilateur **ICC**

Très court régime transitoire, on conserve le nombre de 20 warmups.

`Optimisation O3 xHost` Nous trouvons ce résultat pour L1 :

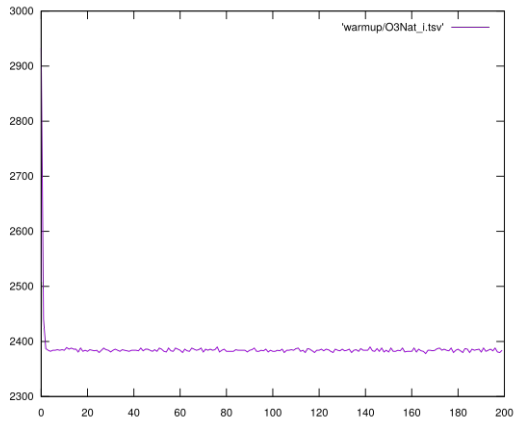

FIGURE 8 – Résultat du warmup avec option `O3 xHost` du compilateur **ICC**

Très court régime transitoire, on conserve le nombre de 20 warmups.

`Optimisation Ofast` Nous trouvons ce résultat pour L1 :

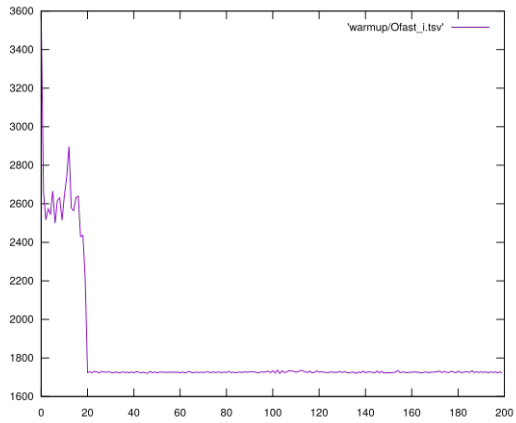

FIGURE 9 – Résultat du warmup avec option `Ofast` du compilateur **ICC**

Ici, le résultat est différent des cas prédédents : on observe que pendant les 20 premiers warmups on est en régime transitoire et qu'on se stabilise au delà. On prendra donc également 20 warmups.

Rétrospectivement, étant donné le nombre très faible de warmups nécessaires à chaque configuration d'optimisation (3 ou 20 ne change pas grand chose, on reste dans le même ordre de grandeur), et par souci de simplicité, nous avons arbitrairement fixé à 20 le nombre de warmups pour toutes les options, au regard de celle étudiée dernièrement et en nécessitant plus que les autres.

II.2.2 L2 : taille 170, L3 : taille 1000, RAM : taille 20000

Les résultats obtenus sont très similaires, on considèrera donc pour chaque optimisation, indépendamment de la taille des données traitées par le noyau, que 20 warmups sont nécessaires à la première méta-répétition, puis on abaissera à 3 warmups pour les méta-répétions suivantes, car la machine est "chaude".

III Résultats et interprétations

III.1 Compilateur GCC©

III.1.1 Option O2

III.1.1.1 Assembleur

```
bcb: 75 f3                  jne    bc0 <baseline+0x30>
bcd: 83 c1 01               add    $0x1,%ecx
bd0: 4c 01 c6               add    %r8,%rsi
bd3: 39 cf                  cmp    %ecx,%edi
bd5: 75 d9                  jne    bb0 <baseline+0x20>
```

III.1.1.2 MAQAO

```
Vectorization
-------------
Your loop is NOT VECTORIZED and could benefit from vectorization.
By vectorizing your loop, you can lower the cost of an iteration
from 4.00 to 1.00 cycles (4.00x speedup).
Since your execution units are vector units,
only a vectorized loop can use their full power.
```

III.1.2 Option O3

III.1.2.1 Assembleur

```
bcb: 75 f3                  jne    bc0 <baseline+0x30>
bcd: 83 c1 01               add    $0x1,%ecx
bd0: 4c 01 c6               add    %r8,%rsi
bd3: 39 cf                  cmp    %ecx,%edi
bd5: 75 d9                  jne    bb0 <baseline+0x20>
```

III.1.2.2 MAQAO

```
Vectorization
-------------
Your loop is NOT VECTORIZED and could benefit from vectorization.
By vectorizing your loop, you can lower the cost of an iteration
from 4.00 to 1.00 cycles (4.00x speedup).
Since your execution units are vector units,
only a vectorized loop can use their full power.
```

III.1.3 Option O3Nat

III.1.3.1 Assembleur

```
bcb: 75 f3                jne    bc0 <baseline+0x30>
bcd: 83 c1 01             add    $0x1,%ecx
bd0: 4c 01 c6             add    %r8,%rsi
bd3: 39 cf                cmp    %ecx,%edi
bd5: 75 d9                jne    bb0 <baseline+0x20>
```

III.1.3.2 MAQAO

```
Vectorization
-------------
Your loop is NOT VECTORIZED and could benefit from
vectorization. By vectorizing your loop, you can lower
the cost of an iteration from 4.00 to 1.00 cycles
(4.00x speedup). Since your execution units are vector
units, only a vectorized loop can use their full power.
```

III.1.4 Option `Ofast`

III.1.4.1 Assembleur

```
bc9: e9 f0 00 00 00        jmpq   cbe <baseline+0x12e>
bce: 66 90                 xchg   %ax,%ax
bd0: 89 f8                 mov    %edi,%eax
bd2: c5 fb 58 02           vaddsd (%rdx),%xmm0,%xmm0
bd6: 83 f8 01              cmp    $0x1,%eax
bd9: 0f 84 51 01 00 00     je     d30 <baseline+0x1a0>
```

III.1.4.2 MAQAO

```
Vectorization status
--------------------
Your loop is fully vectorized (all SSE/AVX instructions
are used in vector version (process two or more data elements
in vector registers), using full register length).
```

III.1.4.3 Justification

`Ofast` Grâce à `ffast-math` le compilateur ignore les normes et la sécurité, en effet ce flag (qui est pratiquement le seul rajouté par le flag `Ofast` à `O3`) active les flags `fno-math-errno, -funsafe-math-optimizations, -fno-trapping-math, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans`. Donc c'est pour cela qu'on arrive à vectorizer notre boucle.

`funroll-loops` Nous remarquons que le code assembleur est différent mais la vectorisation ne change pas, puisque la boucle est déjà complètement vectorisé par Ofast

III.1.4.4 Optimisation `funroll-loops`

Le résultat que nous trouvons est :

Assembleur

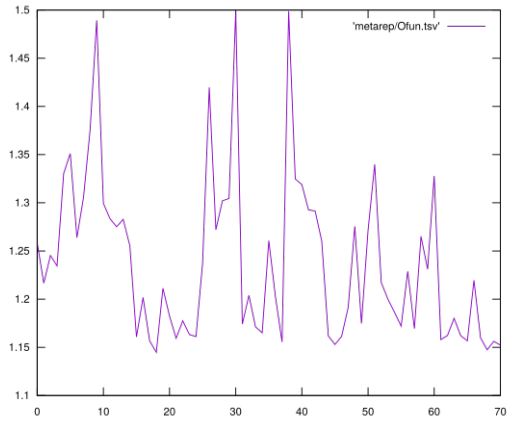

FIGURE 10 – Résultat des métarépétitions avec option `funroll-loops` du compilateur **GCC**©

```
1322: 0f 86 14 01 00 00     jbe    143c <baseline+0x1fc>
1328: 4f 8d 34 f1           lea    (%r9,%r14,8),%r14
132c: 45 8d 78 ff           lea    -0x1(%r8),%r15d
1330: 41 b9 01 00 00 00     mov    $0x1,%r9d
1336: 49 01 f6              add    %rsi,%r14
1339: 41 83 e7 07           and    $0x7,%r15d
133d: c4 c1 7d 28 0e        vmovapd (%r14),%ymm1
1342: 49 8d 46 20           lea    0x20(%r14),%rax
1346: 41 83 f8 01           cmp    $0x1,%r8d
134a: 0f 86 d0 00 00 00     jbe    1420 <baseline+0x1e0>
```

MAQAO

```
Vectorization status
--------------------
Your loop is fully vectorized (all SSE/AVX instructions
are used in vector version (process two or more data elements
in vector registers), using full register length).
```

III.2 Compilateur ICC

III.2.1 Option O2

III.2.1.1 Assembleur

```
400e39: 72 f2                jb     400e2d <baseline+0x5d>
400e3b: 49 63 c0             movslq %r8d,%rax
400e3e: 4d 8d 14 d1          lea    (%r9,%rdx,8),%r10
400e42: 48 83 c2 08          add    $0x8,%rdx
400e46: 66 41 0f 58 1a       addpd  (%r10),%xmm3
400e4b: 66 41 0f 58 52 10    addpd  0x10(%r10),%xmm2
400e51: 66 41 0f 58 4a 20    addpd  0x20(%r10),%xmm1
400e57: 66 41 0f 58 42 30    addpd  0x30(%r10),%xmm0
400e5d: 49 83 c2 40          add    $0x40,%r10
400e61: 48 3b d0             cmp    %rax,%rdx
400e64: 72 dc                jb     400e42 <baseline+0x72>
```

III.2.1.2 MAQAO

```
Vectorization
-------------
Your loop is PARTIALLY VECTORIZED and could benefit from
full vectorization. By fully vectorizing your loop, you can lower
the cost of an iteration from 4.00 to 2.00 cycles (2.00x speedup).
Since your execution units are vector units,
only a fully vectorized loop can use their full power.
```

III.2.2 Option O3

III.2.2.1 Assembleur

```
400e39: 72 f2                jb     400e2d <baseline+0x5d>
400e3b: 49 63 c0             movslq %r8d,%rax
400e3e: 4d 8d 14 d1          lea    (%r9,%rdx,8),%r10
400e42: 48 83 c2 08          add    $0x8,%rdx
400e46: 66 41 0f 58 1a       addpd  (%r10),%xmm3
400e4b: 66 41 0f 58 52 10    addpd  0x10(%r10),%xmm2
400e51: 66 41 0f 58 4a 20    addpd  0x20(%r10),%xmm1
400e57: 66 41 0f 58 42 30    addpd  0x30(%r10),%xmm0
400e5d: 49 83 c2 40          add    $0x40,%r10
400e61: 48 3b d0             cmp    %rax,%rdx
400e64: 72 dc                jb     400e42 <baseline+0x72>
```

III.2.2.2 MAQAO

```
Vectorization
-------------
Your loop is PARTIALLY VECTORIZED and could benefit from
full vectorization. By fully vectorizing your loop, you can lower
the cost of an iteration from 4.00 to 2.00 cycles (2.00x speedup).
Since your execution units are vector units,
only a fully vectorized loop can use their full power.
```

III.2.3 Option O3Nat

III.2.3.1 Assembleur

```
400ea1: 72 f3                jb     400e96 <baseline+0x76>
400ea3: 48 63 d0             movslq %eax,%rdx
400ea6: c5 e5 58 1c ce       vaddpd (%rsi,%rcx,8),%ymm3,%ymm3
400eab: c5 ed 58 54 ce 20    vaddpd 0x20(%rsi,%rcx,8),%ymm2,%ymm2
400eb1: c5 f5 58 4c ce 40    vaddpd 0x40(%rsi,%rcx,8),%ymm1,%ymm1
400eb7: c5 fd 58 44 ce 60    vaddpd 0x60(%rsi,%rcx,8),%ymm0,%ymm0
400ebd: 48 83 c1 10          add    $0x10,%rcx
400ec1: 48 3b ca             cmp    %rdx,%rcx
400ec4: 72 e0                jb     400ea6 <baseline+0x86>
```

III.2.3.2 MAQAO

```
Vectorization status
--------------------
Your loop is fully vectorized (all SSE/AVX instructions
are used in vector version (process two or more data elements
in vector registers), using full register length).
```

III.2.4 Option Ofast

III.2.4.1 Assembleur

```
  400e39: 72 f2                 jb     400e2d <baseline+0x5d>
  400e3b: 49 63 c0              movslq %r8d,%rax
  400e3e: 4d 8d 14 d1           lea    (%r9,%rdx,8),%r10
  400e42: 48 83 c2 08           add    $0x8,%rdx
  400e46: 66 41 0f 58 1a        addpd  (%r10),%xmm3
  400e4b: 66 41 0f 58 52 10     addpd  0x10(%r10),%xmm2
  400e51: 66 41 0f 58 4a 20     addpd  0x20(%r10),%xmm1
  400e57: 66 41 0f 58 42 30     addpd  0x30(%r10),%xmm0
  400e5d: 49 83 c2 40           add    $0x40,%r10
  400e61: 48 3b d0              cmp    %rax,%rdx
  400e64: 72 dc                 jb     400e42 <baseline+0x72>
```

Ofast inclus moins de vectorisation que `O3 -xHost` donc c'est l'option xHost qui nous permet de passer de addpd vers vaddpd.

III.2.4.2 MAQAO

```
Vectorization
-------------
Your loop is PARTIALLY VECTORIZED and could benefit from full
vectorization. By fully vectorizing your loop, you can lower
the cost of an iteration from 4.00 to 2.00 cycles (2.00x speedup).
Since your execution units are vector units,
only a fully vectorized loop can use their full power.
```

III.2.5 Option funroll-loops

Le résultat que nous trouvons est :

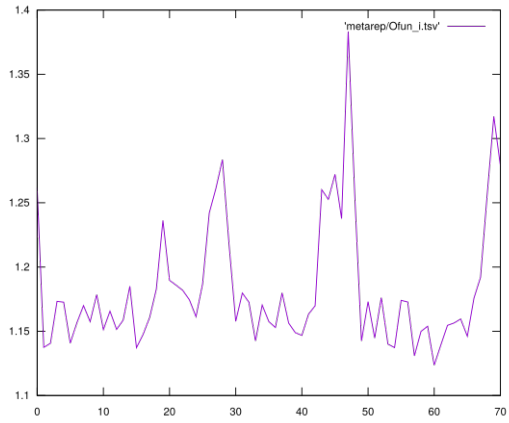

FIGURE 11 – Résultat des métarépétitions avec option `funroll-loops` du compilateur **ICC**

III.2.5.1 Assembleur

```
400ea1: 72 f3                jb     400e96 <baseline+0x76>
400ea3: 48 63 d0             movslq %eax,%rdx
400ea6: c5 e5 58 1c ce       vaddpd (%rsi,%rcx,8),%ymm3,%ymm3
400eab: c5 ed 58 54 ce 20    vaddpd 0x20(%rsi,%rcx,8),%ymm2,%ymm2
400eb1: c5 f5 58 4c ce 40    vaddpd 0x40(%rsi,%rcx,8),%ymm1,%ymm1
400eb7: c5 fd 58 44 ce 60    vaddpd 0x60(%rsi,%rcx,8),%ymm0,%ymm0
400ebd: 48 83 c1 10          add    $0x10,%rcx
400ec1: 48 3b ca             cmp    %rdx,%rcx
400ec4: 72 e0                jb     400ea6 <baseline+0x86>
```

Nous remarquons que contrairement à `Ofast` l'option `funroll-loops` utilise **vaddpd** au lieu de **addpd**, et le résultat de *MAQAO* montre que notre boucle est bel et bien vectorisé

III.2.5.2 MAQAO

```
Vectorization status
--------------------
Your loop is fully vectorized (all SSE/AVX instructions
are used in vector version (process two or more data elements
in vector registers), using full register length).
```

Les médianes des différents temps d'exécution :

GCC ©	`-O2`	3.964161
	`-O3`	3.964172
	`-O3 -march=native`	3.964278
	`-Ofast -march=native`	0.824717
	`-Ofast -march=native -funroll-loops`	1.036789
ICC ©	`-O2`	0.476589
	`-O3`	0.476239
	`-O3 -xHost`	0.645322
	`-Ofast -xHost`	0.476617
	`-Ofast -xHost -funroll-loops`	0.645228