ARM 2 : Etude de cas

RUCKEBUSCH Arnaud

Encadrant : M OSERET Emmanuel

Table des matières

1 Introduction

En programmation informatique, l'optimisation est la pratique qui consiste à réduire le temps d'exécution d'un programme, l'espace occupé par les données ou la consommation d'énergie.

Dans cette étude l'objectif sera d'optimiser une fonction "kernel" et plus particulièrement dans notre cas une fonction utilisant une matrice de taille n*n. Le but sera dans un premier temps d'analyser cette fonction, de produire un driver de test, d'étudier les différentes méthodes d'optimisation pour enfin déterminer des voies d'optimisation de cette fonction.

Je m'intéresse ici plus particulièrement à l'optimisation de la vitesse d'exécution du programme, en utilisant des outils de profiling pour mesurer le temps passé dans les différentes routines du code donné, et ainsi détecter les parties qui prennent un temps anormalement long pour s'exécuter.

2 Environnement d'étude

Pour cette étude les tests sont sur un code en langage C. Différentes optimisations seront réalisées sur deux compilateurs : ICC et GCC. Version 14.0.1 de ICC build le 2013/10/08 et GGC version 4.8.1.

Les tests seront exécutés sur un processeur Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz : Soit une architecture SANDYBRIDGE 64 bits, dans un environnement Linux (Version 13.10 Ubuntu Saucy) en mode FailSafe pour garantir des résultats stables.

Nous nous sommes répartis les différentes études sur les caches et la suite de ce rapport portera sur des tests en cache L2. L'architecture de mes caches se découpe de la façon suivante :

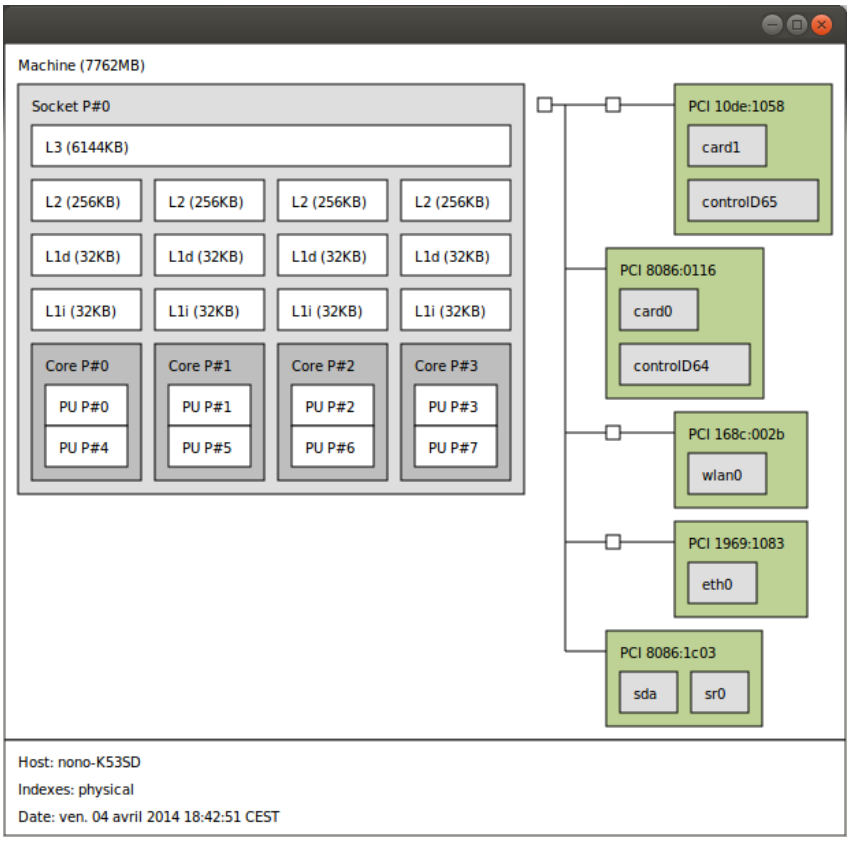

FIGURE 1 – Topologie du système

Donc pour tenir en cache L2 je suis partit sur matrice 200*200. (200*200*4 (étant donné que c'est une matrice de float) = 160 Ko).

3 Partie I - Mesure des options de compilation

3.1 Ecriture du Driver

Pour le driver nous avons repris celui présenté en cours et y avons apporté quelques modifications. Dans un premier temps nous avons rajouté le RDSTC [1] qui va nous permettre de retourner le nombre de ticks écoulés depuis la dernière remise à zéro du processeur. On récupère le rdtsc dans t1 avant l'exécution du kernel et dans t2 après. Size correspond à la taille de la matrice, et rept aux nombres de répétition de la boucle d'exécution du kernel (une centaine de ces itérations suffit). Sur la centaine d'itération il en ressors une médiane qui sera utilisé par un script qui exécutera cette opération 31 fois afin de garantir des résultats stables.

```
    (t2 - t1) / (size * size * rept)
```

Cette formule va nous retourner le nombre de ticks effectué pour une moyenne du temps d'exécution de la fonction kernel en nombre de ticks.

Afin d'avoir toujours le même résultat de calcul du kernel nous effectuons également toujours la même initialisation de notre matrice avec un srand(0).

Pour réaliser les tests de la première partie j'utilise un script générant les différents rapport de tests sur l'affichage des données issues du driver, des rapports maqao et likwid ainsi que du langage assembleur après compilation de la fonction kernel. Afin de ne pas changer les performances du code le driver sera toujours compilé en -O0 afin d'éviter plusieurs problèmes comme la possibilité qu'à le compilateur de ne pas exécuter le kernel si son degré d'optimisation est supérieur à -O2. Le driver, notre fonction kernel ainsi que mes différents scripts de test se trouvent en annexe.

1. Read Time Stamp Counter

3.2 Compilation GCC

Pour chacune des option de compilation je récupère une liste de résultats (chacun de ces résultats correspond à la médiane de 100 itérations de la fonction kernel) de 31 itérations du programme triés qui me permettent de déterminer le min, le max ainsi que la médiane (sur 31 résultats la médiane correspond au 16ème de la liste triée), en vérifiant que le résultat est toujours le bon. Effectuer cette opération me garantit à coût sûr des résultats stables. Les autres rapports me permettent de déterminer les différentes optimisations possibles à faire, ainsi que de comparer les différentes options de compilation.

3.2.1 GCCO2

1. Resultats :

Médiane	Max	Min	Résultat
2.142208	2.211840	2.138112	19921.166016

   Le fait que la médiane et la valeur minimale obtenue lors des tests soit très proches nous indique une bonne stabilité de nos résultats et donc qu'ils sont utilisables.

2. Analyse Maqao :

   "Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode). Only 12% of vector length is used."

   Comme nous le précise la documentation l'option d'optimisation -O2 n'effectue pas de vectorisation et donc il y a une perte de performance.

   "Performance is bounded by DATA DEPENDENCIES."

   Il y a une dépendance de donnée au sein de la boucle étant donné que l'on fait un += à chaque élément de notre matrice et cela porte préjudice à la performance.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00528066
Runtime unhalted [s]	0.00651449
Clock [MHz]	3089.06
CPI	0.612515
L2 request rate	0.0219604
L2 miss rate	0.00074486
L2 miss ratio	0.0339183

Les valeurs les plus importantes à extraire de likwid sont le miss rate et le miss ratio dans le cache L2 que nous allons pouvoir comparer avec les autres options d'optimisation.

4. Analyse des binaires :

Binaires_kernel_GCCO2.txt

```
0000000000400860 <kernel>:
  400860: 85 ff                 test   %edi,%edi
  400862: 0f 57 c0              xorps  %xmm0,%xmm0
  400865: 7e 34                 jle    40089b <kernel+0
     x3b>
  400867: 0f 57 c0              xorps  %xmm0,%xmm0
  40086a: 48 63 cf              movslq %edi,%rcx
  40086d: 48 c1 e1 02           shl    $0x2,%rcx
  400871: 45 31 c0              xor    %r8d,%r8d
  400874: 0f 1f 40 00           nopl   0x0(%rax)
  400878: 48 89 f2              mov    %rsi,%rdx
  40087b: 31 c0                 xor    %eax,%eax
  40087d: 0f 1f 00              nopl   (%rax)
  400880: 83 c0 01              add    $0x1,%eax
  400883: f3 0f 58 02           addss  (%rdx),%xmm0
  400887: 48 01 ca              add    %rcx,%rdx
  40088a: 39 f8                 cmp    %edi,%eax
  40088c: 75 f2                 jne    400880 <kernel+0
     x20>
  40088e: 41 83 c0 01           add    $0x1,%r8d
  400892: 48 83 c6 04           add    $0x4,%rsi
  400896: 41 39 f8              cmp    %edi,%r8d
  400899: 75 dd                 jne    400878 <kernel+0
     x18>
  40089b: f3 c3                 repz retq
  40089d: 0f 1f 00              nopl   (%rax)
```

Le code assembleur est essentiel également. Il va nous permettre en le comparant avec d'autres binaires de voir les réelles optimisations faites par le compilateur. Et pour la seconde partie nous permettre de détecter les voies d'optimisation de manière efficace.

Comme nous le précise MAQAO notre code n'est pas vectorisé. Il est très simple dans ce code assembleur de repérer la double boucle imbriquée avec les deux instructions jne ainsi que l'instruction jle qui correspond à la condition de sortie de boucle principale.

3.2.2 GCCO3

1. Resultats :

Médiane	Max	Min	Résultat
2.150400	2.207744	2.138112	19921.166016

Les résultats sont très proches des précédents. Il va falloir utiliser les outils d'analyse pour déceler des différences si elles existent.

2. Analyse Maqao :

"Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode). Only 12% of vector length is used."

Comme pour O2 ici l'optimisation en O3 n'a pas vectorisé nos boucles.

"Performance is bounded by DATA DEPENDENCIES."

Même problème de dépendances de données qu'avec l'option d'optimisationen O2.

"By removing all these bottlenecks, you can lower the cost of an iteration from 3.00 to 1.33 cycles (2.25x speedup)."

Afin d'augmenter les performances MAQAO nous indique la marche à suivre.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00527399
Runtime unhalted [s]	0.00651916
Clock [MHz]	3091.29
CPI	0.612951
L2 request rate	0.0221234
L2 miss rate	0.000759124
L2 miss ratio	0.0343132

Sur l'analyse en cache L2 celle-ci est similaire à la précédente ? Cela doit signifier qu'il n'y a pas de grandes différences d'optimisation entre les deux options sur ce kernel.

4. Analyse des binaires :

Binaires_kernel_GCCO3.txt

```
0000000000400860 <kernel>:
  400860: 85 ff                 test   %edi,%edi
  400862: 0f 57 c0              xorps  %xmm0,%xmm0
  400865: 7e 34                 jle    40089b <kernel+0
     x3b>
  400867: 0f 57 c0              xorps  %xmm0,%xmm0
  40086a: 48 63 cf              movslq %edi,%rcx
  40086d: 48 c1 e1 02           shl    $0x2,%rcx
  400871: 45 31 c0              xor    %r8d,%r8d
  400874: 0f 1f 40 00           nopl   0x0(%rax)
  400878: 48 89 f2              mov    %rsi,%rdx
  40087b: 31 c0                 xor    %eax,%eax
  40087d: 0f 1f 00              nopl   (%rax)
  400880: 83 c0 01              add    $0x1,%eax
  400883: f3 0f 58 02           addss  (%rdx),%xmm0
  400887: 48 01 ca              add    %rcx,%rdx
  40088a: 39 f8                 cmp    %edi,%eax
  40088c: 75 f2                 jne    400880 <kernel+0
     x20>
  40088e: 41 83 c0 01           add    $0x1,%r8d
  400892: 48 83 c6 04           add    $0x4,%rsi
  400896: 41 39 f8              cmp    %edi,%r8d
  400899: 75 dd                 jne    400878 <kernel+0
     x18>
  40089b: f3 c3                 repz retq
  40089d: 0f 1f 00              nopl   (%rax)
```

Nos doutes sont confirmé par l'analyse des binaires. Les codes assembleur pour les options d'optimisation O2 et O3 sont exacteemnt les mêmes ce qui justifie également les résultats quasiement identiques.

Celà signifie également qu'il y a certaines optimisations qu'effectue GCC O3 et que n'effectue pas GCC O2 qui ne sont pas efficace pour notre kernel.

3.2.3 GCCO3 march=native

1. Resultats :

Médiane	Max	Min	Résultat
2.142208	2.211840	2.134016	19921.166016

2. Analyse Maqao :

   "Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode). Only 12% of vector length is used."

   Comme précédemment la vectorisation n'est pas efficace.

   "Performance is bounded by DATA DEPENDENCIES."

   Soucis identique avec les autres options d'optimisation de GCC.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00530165
Runtime unhalted [s]	0.00652528
Clock [MHz]	3089.2
CPI	0.61353
L2 request rate	0.0221067
L2 miss rate	0.000762122
L2 miss ratio	0.0344747

   Les valeurs sont sensiblement proche avec les autres optimisations.

4. Analyse des binaires :

Binaires_kernel_GCCO3_march.txt

```
0000000000400860 <kernel>:
  400860: 85 ff                 test   %edi,%edi
  400862: c5 f8 57 c0           vxorps %xmm0,%xmm0,%xmm0
  400866: 7e 3b                 jle    4008a3 <kernel+0
     x43>
```

```
400868: 48 63 cf              movslq %edi,%rcx
40086b: 45 31 c0              xor    %r8d,%r8d
40086e: c5 f8 57 c0           vxorps %xmm0,%xmm0,%xmm0
400872: 48 c1 e1 02           shl    $0x2,%rcx
400876: 66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax
   ,1)
40087d: 00 00 00
400880: 48 89 f2              mov    %rsi,%rdx
400883: 31 c0                 xor    %eax,%eax
400885: 0f 1f 00              nopl   (%rax)
400888: 83 c0 01              add    $0x1,%eax
40088b: c5 fa 58 02           vaddss (%rdx),%xmm0,%
   xmm0
40088f: 48 01 ca              add    %rcx,%rdx
400892: 39 f8                 cmp    %edi,%eax
400894: 75 f2                 jne    400888 <kernel+0
   x28>
400896: 41 83 c0 01           add    $0x1,%r8d
40089a: 48 83 c6 04           add    $0x4,%rsi
40089e: 41 39 f8              cmp    %edi,%r8d
4008a1: 75 dd                 jne    400880 <kernel+0
   x20>
4008a3: f3 c3                 repz retq
4008a5: 66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax
   ,1)
4008ac: 00 00 00
4008af: 90                    nop
```

Ce code binaire est légèrement différent de celui de GCC O3, en effet l'addition s'effectue avec un vaddss plutôt qu'un addss (soit en flottant plutôt qu'en entier).

3.2.4 Comparatif

Nous pouvons nous rendre compte que pour ce code il n'y a pas d'optimisation faite entre la compilation en -O2 et la compilation en -O3, en effet lors de l'analyse du code assembleur des deux options d'optimisation ceux-ci sont sensiblement les mêmes. Cependant nous pouvons appercevoir une différence avec l'option march=native qui va inclure des opérations comme le add au milieux de la double boucle en instruction vectorielle, cependant cette légère optimisation tentée par march=native n'est pas concluante et n'offre pas un indice de performance plus élevé qu'en -O2 ou -O3.

-O3 active tous les flags de -O2 en plus des flags suivant : -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload et -ftree-vectorize.

Maqao nous indique quelles sont les voies d'optimisation possible.

3.3 Compilation ICC

3.3.1 ICCO2

1. Resultats :

Médiane	Max	Min	Résultat
0.233472	0.249856	0.229376	19921.269531

Avec ICC on se rend vite compte avec ces premiers résultats que les méthodes d'optimisation ne sont pas les mêmes, bien plus agressif que gcc. Nous pouvons également remarquer que cela affecte un peu le résultat.

2. Analyse Maqao :

"Your loop is vectorized (all SSE/AVX instructions are used in vector mode) but on 50% vector length."

Ici une vrai vectorisation est effectuée, on utilise 50% de la taille de notre vecteur ce qui est déjà bien mieux que les 12% avec GCC.

"By fully vectorizing your loop, you can lower the cost of an iteration from 8.00 to 4.00 cycles (2.00x speedup).
Propositions :
- Pass to your compiler a micro-architecture specialization option :
Intel : use axHost or xHost."

Maqao nous conseille d'utiliser l'optimisation xHost que nous verrons plus tard.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00179418
Runtime unhalted [s]	0.00158524
Clock [MHz]	3086.79
CPI	0.617816
L2 request rate	0.0911208
L2 miss rate	0.00385255
L2 miss ratio	0.0422796

Le fait de vectoriser les données et de régler certains des problème de dépendance de données (notamment en déroulant la boucle) permet d'apporter de bonnes performances sur les taux de miss dans les caches.

4. Analyse des binaires :

Binaires_kernel_ICCO2.txt

```
0000000000400860 <kernel>:
  400860: 33 c0                 xor    %eax,%eax
  400862: 48 63 d7              movslq %edi,%rdx
  400865: 66 0f ef ff           pxor   %xmm7,%xmm7
  400869: 66 45 0f ef c0        pxor   %xmm8,%xmm8
  40086e: 0f 28 f7              movaps %xmm7,%xmm6
  400871: 0f 28 ef              movaps %xmm7,%xmm5
  400874: 0f 28 e7              movaps %xmm7,%xmm4
  400877: 0f 28 df              movaps %xmm7,%xmm3
  40087a: 0f 28 d7              movaps %xmm7,%xmm2
  40087d: 0f 28 cf              movaps %xmm7,%xmm1
  400880: 0f 28 c7              movaps %xmm7,%xmm0
  400883: 48 85 d2              test   %rdx,%rdx
  400886: 0f 8e 2f 01 00 00     jle    4009bb <kernel+0
     x15b>
  ...
```

Ici on se rend compte que le code assembleur est bien plus que lors de la compilation avec GCC. L'ajout d'instructions vectorielles ainsi qu'un léger loop unrolling permet d'améliorer considérablement la performance du code.

Cependant l'agressivité de ICC par rapport à GCC une légère erreur de calcul sur la somme de la matrice.

3.3.2 ICCO3

1. Resultats :

Médiane	Max	Min	Résultat
0.233472	0.258048	0.225280	19921.269531

2. Analyse Maqao :

   Vectorization : Your loop is vectorized (all SSE/AVX instructions are used in vector mode) but on 50% vector length.

   Comme en ICC O2 la vectorisation est plutôt importante.

   Performance is bounded by DATA DEPENDENCIES.

   Le problème de dépendance des données est toujours présent et nous essayerons de le résoudre dans la partie II.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00179332
Runtime unhalted [s]	0.00161621
Clock [MHz]	3088.52
CPI	0.623718
L2 request rate	0.0908461
L2 miss rate	0.00313442
L2 miss ratio	0.0345025

4. Analyse des binaires :

Binaires_kernel_ICCO3.txt

```
0000000000400860 <kernel>:
  400860: 33 c0                 xor    %eax,%eax
  400862: 48 63 d7              movslq %edi,%rdx
  400865: 66 0f ef ff           pxor   %xmm7,%xmm7
  400869: 66 45 0f ef c0        pxor   %xmm8,%xmm8
  40086e: 0f 28 f7              movaps %xmm7,%xmm6
  400871: 0f 28 ef              movaps %xmm7,%xmm5
  400874: 0f 28 e7              movaps %xmm7,%xmm4
  400877: 0f 28 df              movaps %xmm7,%xmm3
```

```
40087a: 0f 28 d7                movaps %xmm7,%xmm2
40087d: 0f 28 cf                movaps %xmm7,%xmm1
400880: 0f 28 c7                movaps %xmm7,%xmm0
...
```

Les résultats sont quasiment les mêmes et cela se perçoit évidement sur le code assembleur qui est sensiblement identique à celui de ICC O2, comme nous l'indique MAQAO, il pourrait être avantageux d'utiliser l'option -xHost qui sera traitée dans la sous section suivante.

3.3.3 ICCO3 xHost

1. Résultats :

Médiane	Max	Min	Résultat
0.278528	0.299008	0.270336	19921.275391

Les résultats sont légèrement moins performant qu'en ICC O3, le résultat de la somme de la matrice est également un petit peut différent.

2. Analyse Maqao :

Your loop is fully vectorized (all SSE/AVX instructions are used in vector mode and on full vector length).

Ici c'est clair, la boucle est complètement vectorisée. Cela affecte comme nous pouvons nous en rendre compte légèrement le résultat de la somme.

Cependant comme précédemment la dépendance de données est toujours bien présente.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00186313
Runtime unhalted [s]	0.0017427
Clock [MHz]	3090.25
CPI	0.631872
L2 request rate	0.0849027
L2 miss rate	0.00271008
L2 miss ratio	0.0319198

Il y a moins de miss dans le cache que en ICC O3.

4. Analyse des binaires :

Binaires_kernel_ICCO3_xHost.txt

```
0000000000400860 <kernel>:
  400860: 33 c0                 xor    %eax,%eax
  400862: 48 63 d7              movslq %edi,%rdx
  400865: c5 e8 57 d2           vxorps %xmm2,%xmm2,%xmm2
  400869: c5 f4 57 c9           vxorps %ymm1,%ymm1,%ymm1
  40086d: c5 fc 28 c1           vmovaps %ymm1,%ymm0
  400871: 48 85 d2              test   %rdx,%rdx
  ...
```

On voit ici que l'option xHost rend le compilateur "plus intelligent" il utilise mieux la vectorisation (en remplissant au mieux les vecteurs)

3.3.4 Comparatif

ICCO2 et ICCO3 sont très agressif sur la vectorisation mais néanmoins efficace, le fait d'utiliser l'option d'optimisation xHost permet de complètement vectoriser le programme. De plus cela apporte encore une légère erreur au résultat.

3.4 Comparaison ICC / GCC

De gros changement s'opèrent entre une compilation avec GCC et ICC. Premièrement ICC est beaucoup plus agressif, il vectorise autant qu'il peut même en O2. Le code binaire généré est bien plus lourd mais également bien plus rapide dû à la vectorisation ainsi qu'au loop unrolling qui reflète certains problèmes de dépendances de données dans la boucle. En terme de performance pure avec ICC nous gagnons quasiment un facteur 10 sur GCC. Cependant la complexité du code assembleur généré par ICC rend son débogage bien plus complexe.

3.5 Autres options d'optimisation

3.5.1 GCC Ofast

L'option d'optimisation Ofast est présente depuis GCC 4.6.

1. Résultats :

Médiane	Max	Min	Résultat
2.129850	2.201200	2.129750	19921.166016

2. Analyse Maqao :

   Les analyses makao sont les même que pour GCC O3.

3. Analyse Likwid :

Metric	core 1
Runtime (RDTSC) [s]	0.00548042
Runtime unhalted [s]	0.00660488
Clock [MHz]	3021.25
CPI	0.614936
L2 request rate	0.0279521
L2 miss rate	0.00625474
L2 miss ratio	0.223766

   Il y a plus de miss dans les caches que en O3.

4. Analyse des binaires :

Binaires_kernel_GCCOfast.txt

```
0000000000400860 <kernel>:
  400860: 85 ff                 test   %edi,%edi
  400862: 0f 57 c0              xorps  %xmm0,%xmm0
  400865: 7e 34                 jle    40089b <kernel+0
      x3b>
  400867: 0f 57 c0              xorps  %xmm0,%xmm0
  40086a: 48 63 cf              movslq %edi,%rcx
  40086d: 48 c1 e1 02           shl    $0x2,%rcx
  400871: 45 31 c0              xor    %r8d,%r8d
  400874: 0f 1f 40 00           nopl   0x0(%rax)
  400878: 48 89 f2              mov    %rsi,%rdx
  40087b: 31 c0                 xor    %eax,%eax
  40087d: 0f 1f 00              nopl   (%rax)
  400880: 83 c0 01              add    $0x1,%eax
  400883: f3 0f 58 02           addss  (%rdx),%xmm0
  400887: 48 01 ca              add    %rcx,%rdx
  40088a: 39 f8                 cmp    %edi,%eax
  40088c: 75 f2                 jne    400880 <kernel+0
      x20>
  40088e: 41 83 c0 01           add    $0x1,%r8d
  400892: 48 83 c6 04           add    $0x4,%rsi
  400896: 41 39 f8              cmp    %edi,%r8d
  400899: 75 dd                 jne    400878 <kernel+0
      x18>
  40089b: f3 c3                 repz retq
  40089d: 0f 1f 00              nopl   (%rax)
```

Avec l'analyse des binaires nous nous rendons compte que celui-ci est identique à celui de O3, ce qui explique les résultats similaires.

3.5.2 ICC Ofast

1. Resultats :

Médiane	Max	Min	Résultat
0.278528	0.299008	0.270336	19921.275391

2. Analyse des binaires :

Binaires_kernel_ICCOfast.txt

```
0000000000400860 <kernel>:
  400860: 33 c0                 xor    %eax,%eax
  400862: 48 63 d7              movslq %edi,%rdx
  400865: 66 0f ef ff           pxor   %xmm7,%xmm7
  400869: 66 45 0f ef c0        pxor   %xmm8,%xmm8
  40086e: 0f 28 f7              movaps %xmm7,%xmm6
  400871: 0f 28 ef              movaps %xmm7,%xmm5
  400874: 0f 28 e7              movaps %xmm7,%xmm4
  400877: 0f 28 df              movaps %xmm7,%xmm3
  40087a: 0f 28 d7              movaps %xmm7,%xmm2
  40087d: 0f 28 cf              movaps %xmm7,%xmm1
  400880: 0f 28 c7              movaps %xmm7,%xmm0
  400883: 48 85 d2              test   %rdx,%rdx
  ...
```

La aussi ICC Ofast est tout aussi efficace que ICC O3

3.6 Option sur -O2

L' option d'optimisation sur -O2 qui nous permettraient de simuler un légère vectorisation est -ftree-vectorize, c'est cette option qu nous permet d'avoir des différences entre GCC -O2 et GCC -O3

4 Partie II - Optimisation du noyau

4.1 Mesure des performances du noyau

Pour cette section nous reprenons les résultats calculés lors de la partie I en GCC O2.

4.2 Limitation des performances

Les performances du code actuellement sont limités par (comme nous en informe makao) la dépendance des données ainsi que la vectorisation. D'après les analyses Likwid nous pouvons voir qu'il serait possible de réduire les taux de miss ce qui pourrait potentiellement améliorer nos performances également.

4.3 Propositions d'optimisations

Dans cette partie nous allons voir de manière itérative différentes option sur le code. Je m'occupe personnellement des effets sur le cache L2 mais il sera intéressant de comparer ces performances avec les effets en RAM [2] ou en L1.

4.3.1 OPTI1

kernel_opti1.c

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

Une des premières option qui nous est venu à l'esprit est de mettre la double boucle dans le bon ordre, en effet afin d'éviter un nombre de miss trop important dans le cache il faut parcourir la matrice par ligne et non par colonnes, ce qui nous permet de ne faire qu'un seul miss par ligne. Cela va se refléter très nettement dans les taux de miss de l'analyse lykwid.

Médiane	Max	Min	Résultat
2.129750	2.200950	2.129700	19921.187500

La valeur de la somme est légèrement différente de celle du kernel original, cela s'explique puisque l'on change le sens de parcours de la matrice et comme les additions ne sont pas commutatives de légères différences apparaissent.

L'analyse maqao ne nous donne pas plus d'informations que sur le kernel de base, ic c'est plus les résultats de likwid qui vont nous intéresser.

2. Random Access Memory

Metric	core 1
Runtime (RDTSC) [s]	0.00538603
Runtime unhalted [s]	0.0066125
Clock [MHz]	3088.14
CPI	0.745256
L2 request rate	0.025496
L2 miss rate	0.000816359
L2 miss ratio	0.0320191

En cache L2 nous n'obtenons pas les résultats escomptés, en effet nous sommes très proche de ce que l'on trouve pour la version originale du kernel. Cependant celà s'explique très bien, en effet, entre ij et ji, nous ne sommes pas avantagés sur les caches car si les données tiennent dans le cache le fait de parcourir la matrice par colonne ou par ligne ne va pas augmenter le taux de miss. Cependant c'est une tout autre histoire dans la RAM c'est le prefetch (prédiction de branchement) suivant qui "fail" si les données ne sont pas contigus et qui nous permet d'avoir une bonne optimisation. (facteur 8 environ en RAM)

4.3.2 OPTI2

kernel_opti2.c

```
float kernel( int n , float a[n][n]){
  int i , j, k ;
  int sizeSum = 6;
  float s[sizeSum];
  float somme = 0.0;
  for( i=0; i < sizeSum ; i++) {
    s[i] = 0.0;
  }

  for ( j = 0; j < n ; j ++){
    i = 0;
    while(i < (n - (n%sizeSum))) {
      for( k=0; k < sizeSum ; k++) {
        s [k] += a [ j ][ i ];
        i++;
      }
    }
    for(k=n - (n%sizeSum); k < n; k++) {
      somme += a [ j ][ k ];
    }
  }
```

```
    for( i=0; i < sizeSum ; i++) {
      somme += s[i];
    }
    return somme ;
  }
```

Pour cette seconde option nous allons faire une sorte de loop unroll afin d'éliminer au maximum la dépendance de donnée.

Médiane	Max	Min	Résultat
1.724850	1.809650	1.722800	19921.281250

Nous avons fait plusieurs tests (de 4 à 8) et nous nous sommes aperçu que c'est avec une valeur de 6 (valeur de découpage) que nous obtenons les meilleurs performances. Ici par rapport au kernel original nous gagnons environ 20% de performances.

Les analyses maqao ne nous donne pas énormément d'information utile, puisqu'un dépendance des données persiste toujours. Likwid nous permet de voir que le taux de miss est plus élevé que pour la précédente optimisation.

Binaires_kernel_GCCO2_opti2.txt

```
00000000004009c0 <kernel>:
  4009c0: 55                    push   %rbp
  4009c1: 0f 57 c0              xorps  %xmm0,%xmm0
  4009c4: 85 ff                 test   %edi,%edi
  4009c6: 53                    push   %rbx
  4009c7: f3 0f 11 44 24 d8     movss  %xmm0,-0x28(%rsp)
  4009cd: f3 0f 11 44 24 dc     movss  %xmm0,-0x24(%rsp)
  4009d3: f3 0f 11 44 24 e0     movss  %xmm0,-0x20(%rsp)
  4009d9: f3 0f 11 44 24 e4     movss  %xmm0,-0x1c(%rsp)
  4009df: f3 0f 11 44 24 e8     movss  %xmm0,-0x18(%rsp)
  4009e5: f3 0f 11 44 24 ec     movss  %xmm0,-0x14(%rsp)
  4009eb: 0f 8e b0 00 00 00     jle    400aa1 <kernel+0xe1>
  ...
```

Sur les binaires nous apercevons bien les 6 movss correspondant au nombre de fois où nous allons découper notre somme.

4.4 Utilisation Maqao - Likwid

Maqao et likwid nous ont beaucoup aidés dans la réalisation de ces optimisations. Likwid perfctr nous as permis de détecter les cache miss très nombreux et donc de nous diriger vers la première optimisation. Ensuite Maqao cqa nous a indiquer les data depencies ainsi que la vectorisation que nous avons essayés de corriger dans les autres options d'optimisations.

5 Outil externe utilisé

Afin de vérifier que la fréquence de mon CPU (et de mon turbo boost) n'influait pas sur les résultats obtenus j'ai cherché à modifier celle-ci. Voici donc les étapes que j'ai suivit pour y parvenir :

1. Installer les packages :
   sudo apt-get install cpufreq cpufrequtils indicator-cpufreq
   sudo apt-get install gnome-applets

2. Vérifier l'état de la fréquence :
   cpufreq-info -p
   -> retourne la fréquence du processeur max / min ainsi que son mode

3. Lancer l'outil en interface graphique :
   indicator-cpufreq
   Sur ubuntu cela rajoute un petit outils dans la barre des tâche linux en haut a droite, ou apparaît une palette de fréquence et les différents mode. (Cependant cliquer dessus ne marche pas de mon côté, mais utile pour la suite)

4. Desactiver le mode ondemand et activer le mode userspace pour tous les cpu
   sudo update-rc.d ondemand disable
   sudo cpufreq-set -c 0-7 -g userspace

5. Mettre a jour la fréquence du cpu à la fréquence de base :
   sudo cpufreq-set -c 0-7 -d 2200000 -u 2200000
   -d et -u sotn là pour indiquer le min et le max (2200000 = 2.2GHz)
   -c 0-7 pour mes 8 cpu

Malheureusement effectuer cette modification n'a pas apporter de changements significatifs aux résultats.

6 Conclusion

Pour conclure à cette étude nous pouvons dire qu'il existe un très grand nombre d'options d'optimisations possible, que celles-ci doivent être faites en fonction du besoin final : résultat précis ou non - rapidité - utilisation mémoire ... Il est également possible par une simple réflexion d'augmenter drastiquement la performance d'un code, il faut pour cela utiliser intelligemment les outils MAQAO et LIKWID et faire des conclusions sur des résultats probants.

En combinant toutes les optimisations sur GCCO2 que nous avons fait dans la partie II nous nous rendons bien compte qu'il est vitale d'opérer celles-ci.

7 Bibliographie

1. Site de redhat :
   www.redhat.com/magazine/011sep05/features/gcc/
2. Site de Gentoo :
   wiki.gentoo.org/wiki/
3. Site de l'université d'Angers :
   www.info.univ-angers.fr
4. Site de GCC :
   gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

8 Annexe

TEST_DRIVER.sh

```
##################### PARTIE I #####################


##################
######GCC O2######
##################
gcc -c -O2 kernel.c
gcc -O0 test_archi.c kernel.o -o test

./test_generique.sh resultatsGCCO2.txt

##################
######GCC O3######
##################
gcc -c -O2 kernel.c
gcc -O0 test_archi.c kernel.o -o test

./test_generique.sh resultatsGCCO3.txt

##################
###GCC O3 march###
##################
gcc -c -O3 -march=native kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsGCCO3_march.txt

##################
#####GCC Ofast####
##################
gcc -c -Ofast kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsGCCOfast.txt

#Intel Compiler
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/lib/ia32
PATH=$PATH:/opt/intel/bin

##################
######ICC O2######
##################
icc -c -g -O2 kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsICCO2.txt

##################
```

```
######ICC O3######
##################
icc -c -g -O3 kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsICCO3.txt

##################
###ICC O3 xHost###
##################
icc -c -O3 -g -xHost kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsICCO3_xHost.txt

##################
####ICC 0fast#####
##################
icc -c -g -Ofast kernel.c
gcc -O0 test_archi.c kernel.o -o test
./test_generique.sh resultatsICCOfast.txt


###################### PARTIE II ######################

##################
#####OPTI 1#######
##################
gcc -c -O2 kernel_opti1.c
gcc -O0 test_archi.c kernel_opti1.o -o test

./test_generique.sh resultatsGCCO2_opti1.txt

##################
#####OPTI 2#######
##################

gcc -c -O2 kernel_opti2.c
gcc -O0 test_archi.c kernel_opti2.o -o test

./test_generique.sh resultatsGCCO2_opti2.txt
```

Script lancé en mode fail safe pour exécuter tous les options d'optimisations.

test_generique.sh

```
# Meta-repetitions
if [ -e $1 ]; then
  rm -f $1
fi

for i in $(seq 1 31)
do
  ./test 200 100 >> tri_des_resultats_tmp.txt
done

#TRI DES RESULTATS

sort tri_des_resultats_tmp.txt > $1
rm -f tri_des_resultats_tmp.txt

#MAQAO

echo "################################
######### ANALYSE MAQAO #######
################################" >> $1

./maqao cqa ./test fct=kernel uarch=SANDY_BRIDGE >> $1

#LIKWID

sudo modprobe msr
sudo chmod 777 /dev/cpu/*/msr

echo "################################
######## ANALYSE LIKWID #######
################################" >> $1

sudo likwid-perfctr -C 1 -g L2CACHE ./test 200 100 >> $1

echo "################################
######## ANALYSE BINAIRE #######
################################" >> $1

objdump -d test >> $1
```

Permet de trier la liste des éléments et de générer les rapports makao et likwid dans le même fichier.

kernel.c

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

Fonction kernel de base à optimiser.

kernel_opti1.c

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

Première optimisation tenté sur la fonction kernel.

kernel_opti2.c

```
float kernel( int n , float a[n][n]){
  int i , j, k ;
  int sizeSum = 6;
  float s[sizeSum];
  float somme = 0.0;
  for( i=0; i < sizeSum ; i++) {
    s[i] = 0.0;
  }

  for ( j = 0; j < n ; j ++){
    i = 0;
    while(i < (n - (n%sizeSum))) {
      for( k=0; k < sizeSum ; k++) {
        s [k] += a [ j ][ i ];
        i++;
      }
    }
    for(k=n - (n%sizeSum); k < n; k++) {
      somme += a [ j ][ k ];
    }
  }
  for( i=0; i < sizeSum ; i++) {
    somme += s[i];
  }
  return somme ;
}
```

Seconde optimisation tenté sur la fonction kernel.

test_archi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

float kernel( int n , float a[n][n]);

/* READ TIME STAMP */
uint64_t rdtsc(void) {
  uint64_t a, d;
  __asm__ volatile ("rdtsc" : "=a" (a), "=d" (d));
  return (d<<32) | a;
}

float initialize(int size, float a[size][size]) {
  int i,j;
  for ( i = 0; i < size ; i ++){
    for ( j = 0; j < size ; j ++){
      a [ i ][ j ] = (float) rand()/RAND_MAX;
    }
  }
}

int main (int argc, char *argv[]) {
  int r;
  int size = atoi(argv[1]);
  int rept = atoi(argv[2]);
  srand(0);

    float *a = malloc(size * size * sizeof *a);

  initialize(size,(float (*)[size]) a);

  printf("%f",kernel(size, (float (*)[size]) a));

  uint64_t t1 = rdtsc();
  for (r=0; r<rept; r++){
    kernel(size, (float (*)[size]) a);
  }
  uint64_t t2 = rdtsc();

  printf ("\t %.6f \n",(float) ((float)((float)t2 - t1) / (
      float)((float)size * size * rept)));
  return 0;
}
```

Driver de test rédigé sur la base de celui donnée en cours.