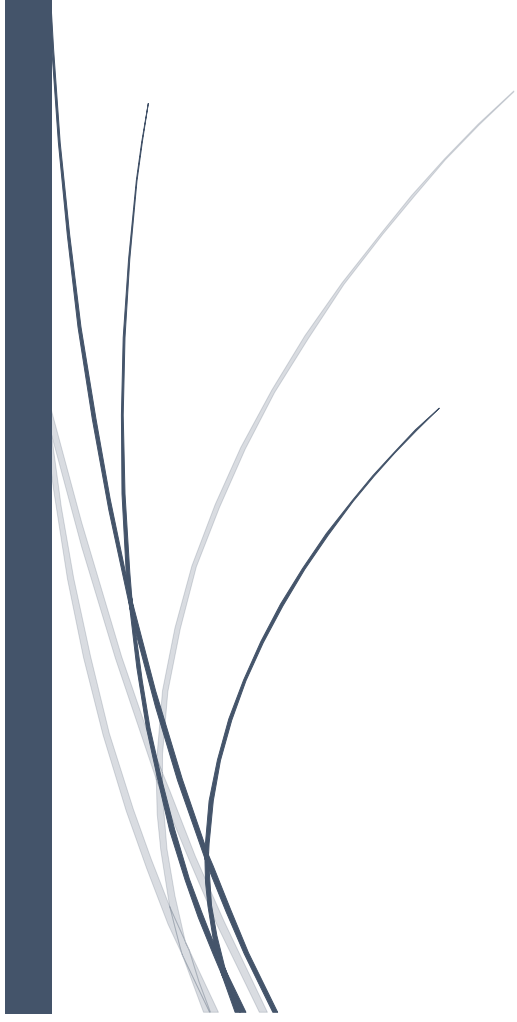
Rapport d'étude de cas d'architecture

Philippe Joulot



Groupe 4 (Sujet 5)

Joulot Philippe – Norbal Xavier – Ruckebusch Arnaud

TABLE DES MATIERES

INTRODUCTION 2
ENVIRONNEMENT DE TRAVAIL 2
Logiciels 2
Architecture 2
Choix de la taille de la matrice 4
PARTIE 1 – DRIVER ET OPTIONS DE COMPILATION 4
Implémentation du driver 4
Le noyau avec gcc -O2 5
Mesures avec gcc -O3, gcc -O3 -march=native, icc -O2, icc -O3 et icc -O3 -xHost 5
Analyse des valeurs de la somme 6
Analyse des temps 6
Autres options d'optimisations 9
Options pour GCC 9
Options pour ICC 10
PARTIE 2 – OPTIMISATIONS DU NOYAU EN GCC –O2 11
Préambule 11
Mode texte 12
Inversion des boucles 12
Loop unrolling 14
Double parcours de matrice 14
CONCLUSION 14
BIBLIOGRAPHIE 16
ANNEXES 17
Annexe 1 : kernelPartie1.c 17
Annexe 2 : test_archi.c 17
Annexe 3 : kernelPartie2Optim1.c 18
Annexe 4 : kernelPartie2Optim2.c 19
Annexe 5 : KERNELPARTIE2OPTIM3.c 20
Annexe 6 : Rapport maqao du noyau compilé en gcc-O2 20

INTRODUCTION

Dans le cadre du cours d'architecture, nous avons réalisé une étude de performances sur un noyau qui nous a été fourni et qui est disponible en annexe 1. Ce noyau prend en entrée une matrice et renvoie la valeur correspondant à la somme de tous les éléments de cette matrice. Le but de cette étude est d'étudier les performances réalisées par le programme et de réfléchir aux différents moyens permettant d'améliorer celles-ci. Un « niveau » d'étude a été désigné pour chacun des membres de notre groupe. Celui qui m'a été incombé est la RAM.

Nous évaluerons tout d'abord les performances de ce noyau au travers de différents compilateurs et de leurs options respectives. Puis nous nous intéresserons à la façon dont on peut modifier le code du noyau afin d'optimiser les performances en utilisant uniquement gcc -O2. Enfin nous essaierons d'améliorer encore le temps d'exécution en se soustrayant de cette contrainte du compilateur et de son option.

ENVIRONNEMENT DE TRAVAIL

LOGICIELS

Pour la version d'ICC, il s'agit de la version 14.0.1 de numéro de build 20131008.

Pour la version de GCC, il s'agit de la version gcc (Ubuntu/Linaro 4.8.1-10ubuntu9) 4.8.1

Pour maqao, on utilise la version 2.1.1 de numéro de build 20131120

En ce qui concerne likwid, j'ai installé chaque version existante afin de tester si l'option MEM fonctionnait. Malheureusement aucune ne fonctionnait car mon processeur n'était pas supporté.

ARCHITECTURE

En utilisant la commande cat /proc/cpuinfo, on obtient les informations détaillées du processeur sur lequel nous allons effectuer les tests. En voici le résultat :

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
stepping        : 9
microcode       : 0x12
cpu MHz                 : 1200.000
cache size      : 6144 KB
physical id     : 0
siblings        : 8
core id         : 0
```

cpu cores : 4

apicid : 0

initial apicid : 0

fpu : yes

fpu_exception : yes

cpuid level : 13

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms

bogomips : 4789.19

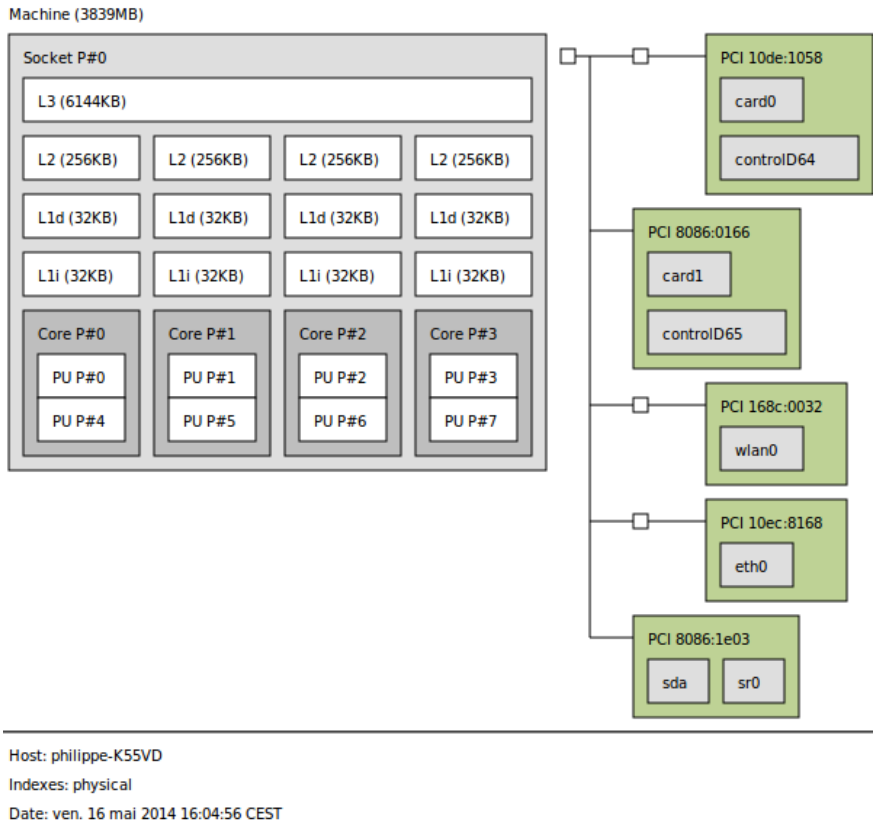
clflush size : 64

cache_alignment : 64

address sizes : 36 bits physical, 48 bits virtual

power management:

Grâce à la commande lstopo, on obtient également l'architecture des caches sous forme de schéma :



CHOIX DE LA TAILLE DE LA MATRICE

Afin de réaliser des mesures, il faut en premier lieu définir la taille de la matrice que l'on utilisera.

Même si certaines parties des caches ne sont pas utilisables (en particulier au niveau L3 où on peut compter sur environ la moitié), j'ai décidé de choisir une taille supérieure à la somme des tailles des caches.

L3 + L2 + L1 = 6144KB + 256KB + 32KB = 6432KB

Sachant qu'un float fait 4 octets, 6432KB correspond à 1 608 000 éléments.

J'ai choisi une taille 2000 pour ma matrice ce qui fait 2 000 000 éléments soit 8 000KB.

PARTIE 1 – DRIVER ET OPTIONS DE COMPILATION

IMPLEMENTATION DU DRIVER

Pour commencer, nous avons recopié comme il nous a été spécifié la fonction RDTSC qui va nous renvoyer le nombre de ticks qui s'est écoulé depuis la dernière remise à zéro du processeur. Ce nombre de ticks va ainsi nous permettre de mesurer le temps pris par notre noyau et sera donc à la base de tous nos calculs.

Ensuite on récupère en arguments du programme, la taille de la matrice voulue ainsi que le nombre d'itérations à réaliser.

A partir de cela, on peut déclarer notre matrice et l'initialiser. On a pris soin d'utiliser la fonction srand(0); afin qu'on obtienne toujours les mêmes valeurs d'initialisation dans notre matrice. Cette précaution est absolument nécessaire pour garantir que les mêmes calculs sont effectués à chaque exécution et donc permettre une comparaison correcte.

On effectue ensuite un warmup en exécutant une fois notre noyau puis on le réexécute autant de fois qu'on a passé de répétitions en argument du programme.

Lors de ces répétitions, on va calculer notre temps suivant la formule :

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

où t2 – t1 représente le nombre de cycles processeur passé dans le kernel

où size * size représente la taille de notre matrice

et où rept représente le nombre d'itérations effectuées.

On va garder le minimum, le maximum, la moyenne ainsi que la médiane afin de pouvoir réaliser notre étude.

L’implémentation du driver est disponible en annexe 2.

LE NOYAU AVEC GCC -O2

Après avoir compilé le noyau en gcc -O2, on obtient les temps d'exécutions suivants :

```
################################
Partie1 - GCC 02
################################
Valeur s = 1999849.625000
moy = 21.082584
min = 20.395555
max = 22.153416
med = 21.087709
```

La valeur s correspond à la somme des éléments de la matrice.

MESURES AVEC GCC -O3, GCC -O3 -MARCH=NATIVE, ICC -O2, ICC -O3 ET ICC -O3 -XHOST

Voici désormais les mesures effectuées avec les autres compilateurs demandés.

```
################################
Partie1 - GCC 03
################################
Valeur s = 1999849.625000
moy = 21.696301
min = 20.969023
max = 22.286627
med = 21.711411
################################
Partie1 - GCC 03 march
################################
Valeur s = 1999849.625000
moy = 21.599199
min = 20.375511
max = 21.999762
med = 21.631483
################################
Partie1 - ICC 02
################################
```

```
Valeur s = 1999882.000000
moy = 0.744833
min = 0.733896
max = 1.408507
med = 0.738070
################################
Partie1 - ICC 03
################################
Valeur s = 1999882.000000
moy = 0.746566
min = 0.735110
max = 1.446540
med = 0.739231
################################
Partie1 - ICC 03xHost
################################
Valeur s = 1999881.000000
moy = 0.780629
min = 0.767506
max = 1.647802
med = 0.772531
```

ANALYSE DES VALEURS DE LA SOMME

En premier lieu on peut constater une différence au niveau de la valeur de la somme s des éléments de la matrice. On constate que gcc et icc ne doivent pas effectuer le calcul de la même façon puisqu'il y a une différence d'environ 0,56 sur la valeur de s, ICC 03xHost mis à part.

Pour icc -O3 -xHost, on remarque une légère différence avec les autres compilations d'icc effectuées ici. L'option -xHost est équivalente à l'option -xCORE-AVX2 ; cela génère un code assembleur spécifique pour les processeurs intel. Le programme ne s'exécutera pas sur les machines possédant des processeurs qui ne sont pas de cette marque. De plus, il faut que le processeur supporte les instructions Intel® AVX2 puisque c'est ce jeu d'instructions qui est utilisé.

ANALYSE DES TEMPS

On observe des temps similaires entre gcc -O2 et gcc -O3. Afin de comprendre pourquoi, je suis allé voir dans le code assembleur des deux kernels compilés. On s'aperçoit qu'il s'agit en fait du même code assembleur :

```
0000000000000000 <kernel>:
   0:   85 ff              test   %edi,%edi
   2:   0f 57 c0                   xorps  %xmm0,%xmm0
   5:   7e 34              jle    3b <kernel+0x3b>
   7:   0f 57 c0                   xorps  %xmm0,%xmm0
   a:   48 63 cf                   movslq %edi,%rcx
   d:   48 c1 e1 02                shl    $0x2,%rcx
  11:   45 31 c0                   xor    %r8d,%r8d
  14:   0f 1f 40 00                nopl   0x0(%rax)
  18:   48 89 f2                   mov    %rsi,%rdx
  1b:   31 c0              xor    %eax,%eax
  1d:   0f 1f 00                   nopl   (%rax)
  20:   83 c0 01                   add    $0x1,%eax
  23:   f3 0f 58 02                addss  (%rdx),%xmm0
  27:   48 01 ca                   add    %rcx,%rdx
  2a:   39 f8              cmp    %edi,%eax
  2c:   75 f2              jne    20 <kernel+0x20>
  2e:   41 83 c0 01                add    $0x1,%r8d
  32:   48 83 c6 04                add    $0x4,%rsi
  36:   41 39 f8                   cmp    %edi,%r8d
  39:   75 dd                      jne    18 <kernel+0x18>
  3b:   f3 c3              repz retq
```

Cela explique donc totalement cette similarité des temps.

Pour gcc -O3 -march=native, les temps sont également similaires à ses homologues O2 et O3. On regarde donc aussi l'assembleur mais ici on peut noter de légères différences.

```
0000000000000000 <kernel>:
   0:   85 ff              test   %edi,%edi
   2:   c5 f8 57 c0                vxorps %xmm0,%xmm0,%xmm0
   6:   7e 3b              jle    43 <kernel+0x43>
   8:   48 63 cf                   movslq %edi,%rcx
```

```
 b:   45 31 c0              xor    %r8d,%r8d
 e:   c5 f8 57 c0           vxorps %xmm0,%xmm0,%xmm0
12:   48 c1 e1 02           shl    $0x2,%rcx
16:   66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax,1)
1d:   00 00 00
20:   48 89 f2              mov    %rsi,%rdx
23:   31 c0           xor    %eax,%eax
25:   0f 1f 00              nopl   (%rax)
28:   83 c0 01              add    $0x1,%eax
2b:   c5 fa 58 02           vaddss (%rdx),%xmm0,%xmm0
2f:   48 01 ca              add    %rcx,%rdx
32:   39 f8           cmp    %edi,%eax
34:   75 f2           jne    28 <kernel+0x28>
36:   41 83 c0 01           add    $0x1,%r8d
3a:   48 83 c6 04           add    $0x4,%rsi
3e:   41 39 f8              cmp    %edi,%r8d
41:   75 dd                 jne    20 <kernel+0x20>
43:   f3 c3           repz retq
```

Parmi les différences, on peut relever par exemple l'instruction nopw qui a été utilisée dans gcc -O3 -march=native. Celle-ci va réaliser un padding afin que le code commence à une certaine adresse mémoire. C'est effectivement spécifique de l'architecture. Il remplace également des xorps par des vxorps et les addss par des vaddss.

Bien qu'icc -O3 réalise des optimisations plus agressives en ce qui concerne notamment le prefetching, les transformations de boucle et la duplication de code, les codes assembleurs produits par icc – O2 et icc -O3 sont exactement pareils. Cela explique les similarités entre les mesures du nombre de cycles.

Avec icc -O3 -xHost, on a imposé le jeu d'instructions à AVX2. Comme le temps d'exécution a légèrement augmenté, c'est que certainement icc -O3 à choisi par défaut un autre jeu d'instructions plus efficace sur notre code.

AUTRES OPTIONS D'OPTIMISATIONS

On notera tout d'abord que les plus hauts niveaux d'optimisations des compilateurs inclus de très nombreuses options et qu'il est relativement difficile de trouver des options supplémentaires entraînant un gain de performance.

OPTIONS POUR GCC

-OFAST

```
Valeur s = 1999849.625000
moy = 21.361475
min = 19.753345
max = 22.115049
med = 21.416477
```

Malheureusement, en comparant l'assembleur produit avec celui de gcc -O3, on s'apperçoit que le code assembleur généré est le même.

-MSSE3, -MMMX OU -M3DNOW

```
Valeur s = 1999849.625000
moy = 18.850422
min = 18.390631
max = 20.519049
med = 18.806019
```

Les résultats se sont améliorés de manière non négligeable avec cette option. Cette option permet de choisir un jeu d'instructions SSE3, MMX ou M3DNow. Le code assembleur généré est le même pour ces trois options. On peut expliquer le gain de performances par le fait que ces jeux d'instructions sont spécifiques aux architectures x86 et x86-64. De plus, ils utilisant les calculs en virgule flottante de manière intensive. Cela s'inscrit donc bien dans notre calcul de la somme des éléments d'une matrice contenant des floats.

Dans les options brutales déconseillées par le manuel de gcc :

-FFORCE-ADDR

```
Valeur s = 1999849.625000
moy = 18.595955
min = 18.123434
max = 22.813482
med = 18.570263
```

On obtient ici un gain de performance visible cependant cette option peut se révéler être une source de bugs.

OPTIONS POUR ICC

-OFAST OU -O3 -FP-MODEL FAST=1

```
Valeur s = 1999882.000000
moy = 0.771924
min = 0.737226
max = 1.968788
med = 0.758017
```

En regardant l'assembleur, on se rend à nouveau compte qu'il est aussi performant que icc -O3. De plus, on peut noter que l'option -fp-model fast=1 autorise des optimisations agressives pouvant altérer la précision des calculs sur les nombres en virgule flottante.

-O3 -MSSE3

```
Valeur s = 1999882.000000
moy = 0.767860
min = 0.736664
max = 1.724503
med = 0.753684
```

En regardant l'assembleur, on constate quelques changements comme les pxor qui se transforment en xorps mais les performances sont très similaires et on ne distingue pas de réelle amélioration.

```
0000000000000000 <kernel>:
  0:    33 c0             xor    %eax,%eax
  2:    48 63 d7                 movslq %edi,%rdx
```

```
 5:    0f 57 ff          xorps  %xmm7,%xmm7
 8:    45 0f 57 c0             xorps  %xmm8,%xmm8
 c:    0f 28 f7                movaps %xmm7,%xmm6
 f:    0f 28 ef                movaps %xmm7,%xmm5
12:    0f 28 e7                movaps %xmm7,%xmm4
15:    0f 28 df                movaps %xmm7,%xmm3
18:    0f 28 d7                movaps %xmm7,%xmm2
1b:    0f 28 cf                movaps %xmm7,%xmm1
1e:    0f 28 c7                movaps %xmm7,%xmm0
21:    48 85 d2                test   %rdx,%rdx
24:    0f 8e 2e 01 00 00       jle    158 <kernel+0x158>
2a:    45 32 d2                xor    %r10b,%r10b
2d:    83 ff 20                cmp    $0x20,%edi
30:    0f 8c 72 01 00 00       jl     1a8 <kernel+0x1a8>
36:    49 89 f1                mov    %rsi,%r9
39:    49 83 e1 0f             and    $0xf,%r9
...                            ...
```

PARTIE 2 – OPTIMISATIONS DU NOYAU EN GCC –O2

Avant de commencer la présentation des optimisations, je tiens à préciser qu'une démarche itérative a été suivie afin d'améliorer à chaque étape les temps d'exécution. Nous allons ici voir la démarche qui a été suivie au cours de l'étude de cas.

PREAMBULE

Avant de nous lancer dans les optimisations, nous avons réalisé un rapport maqao afin d'observer le speedup que l'on peut obtenir ainsi que des pistes d'optimisations.

Le rapport maqao est disponible en annexe 6.

Comme on peut le voir, la boucle n'est pas vectorisée.

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

Il faudrait donc utiliser une autre option de compilation comme O3 ou un autre compilateur comme icc afin que la boucle soit vectorisée. Mais on est limitée par la contrainte de GCC –O2.

Performance is bounded by DATA DEPENDENCIES.

By removing most critical dependency chains, you can lower the cost of an iteration from 3.00 to 1.00 cycles (3.00x speedup).

Two propositions:

- Try another compiler or update/tune your current one:

- Remove inter-iterations dependences from your loop.

Comme on le voit, le problème principal de ce programme est la dépendance de données inter-itération. En effet, à chaque itération on somme un élément de la matrice. Le parcours se fait donc un élément par un élément.

Les propositions proposées par maqao sont :

- changer de compilateur : ce qui nous est impossible puisqu'on est contraint d'utiliser GCC-O2
- Supprimer la dépendance de données

Nous allons donc désormais voir la démarche des optimisations réalisées afin d'obtenir de meilleures performances.

MODE TEXTE

La première chose que nous avons fait pour améliorer les performances, c'est de lancer nos machines en mode console et non en interface graphique en remplaçant « quiet splash » par « text » dans le fichier de notre grub. Après avoir mis à jour le grub, les résultats obtenus sont alors devenus plus stables car moins de processus venaient « polluer » notre processeur. Par conséquent, on note une amélioration aux niveaux des temps d'exécution du noyau. En voici les résultats :

```
################################
Partie1 - GCC 02
################################
Valeur s = 1999849.625000
moy = 19.507526
min = 19.184443
max = 20.443258
med = 19.477072
```

INVERSION DES BOUCLES

Notre première optimisation a été d'inverser les deux boucles pour le parcours de la matrice.

Comme dans le langage c, les données sont stockées par ligne et non par colonne ; le fait de parcourir la matrice ligne par ligne nous a permis d'améliorer les performances. On obtient un facteur d'amélioration de presque 10.

```
################################
Partie2Optim1 - GCC 02
################################
Valeur s = 1999958.750000
moy = 2.204123
min = 2.156189
max = 2.855870
med = 2.191698
```

En effet, cela s'explique par le fait que les données vont être chargées par lignes dans le cache et cela va nous permettre d'obtenir un miss puis que des hits par ligne de cache.

Auparavant, la version originale du kernel parcourait par colonne donc dans ce cas-là on avait que des miss.

De plus, le prefetch que le système va effectuer pour charger les données à l'avance sera plus efficace puisque les données sont prefetchées par ligne et qu'on lit désormais par ligne. Comme la partie dont je suis chargé est la RAM, les données ne tiennent pas dans les caches et le système de prefetch est donc très efficace comme en témoigne les résultats.

On peut aussi noter que la valeur de la somme a légèrement changée. Ceci est dû au fait que la somme n'est pas associative sur les floats et qu'inverser les boucles a donc changer l'ordre des opérations. Cependant suite à un rendez-vous, cette variation a été considérée comme négligeable et l'optimisation a donc été validée.

Le code de cette optimisation est disponible en annexe 3.

LOOP UNROLLING

Notre deuxième optimisation est basée sur le principe *loop unrolling.* Nous avons créé un tableau de float afin de stockées des sommes partielles. Dans notre boucle la plus interne, on va la dérouler afin d'effectuer plusieurs calculs. Chaque opération add déroulé s'affectera sur une somme partielle.

A la fin du parcours de la matrice, on va simplement faire la somme des sommes partielles afin d'obtenir notre valeur finale.

```
################################
Partie2Optim2 - GCC 02
################################
Valeur s = 1999896.375000
moy = 1.650795
min = 1.598067
max = 2.786179
med = 1.635972
```

On obtient ici une amélioration des performances, cela s'explique notamment par le fait que l'instruction de saut de la boucle est beaucoup moins exécutée.

Le code de cette optimisation est disponible en annexe 4.

DOUBLE PARCOURS DE MATRICE

Nous avions tenté une troisième optimisation en essayant de parcourir à la fois la première ligne de notre matrice et la dernière. Un parcours simultané en partant de la fin et du début. Après évaluation des performances, on s'est rendu compte que celles-ci s'étaient dégradées. En ajoutant cette idée, le code écrit s'est révélé beaucoup plus complexe et c'est cela qui explique très certainement cette dégradation de performances.

Le code de cette « optimisation » est disponible en annexe 5.

CONCLUSION

Cette étude de cas nous a permis d'explorer les différentes options de compilation des compilateurs gcc et icc et personnellement cela me sera très utile pour mes futurs programmes afin d'optimiser les performances facilement sans avoir besoin de faire des modifications. En ayant fait des recherches, je suis également tombé sur des options pour réduire la taille du binaire ou être plus économe en énergie, ce qui peut s'avérer également utile.

De plus, la partie 2 nous a permis de s'interroger sur du code et de chercher parmi des optimisations au niveau source et de trouver lesquelles seraient les plus adaptées et les plus pertinentes afin d'obtenir de meilleures performances. Ce travail de recherche entrainant inévitablement une liaison avec les cours magistraux d'architecture, m'a permis de pouvoir appliquer en conditions réelles les principes vu en architecture et de pouvoir constater des résultats. Enfin, je conclurai sur le fait qu'il est important de bien penser son code source puisqu'on peut obtenir des facteurs de performance bien meilleurs comme on a pu le constaté au cours de cette étude.

BIBLIOGRAPHIE

- **Site officiel de GCC**
  http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- **Site de Golden Energy Computing Organization**
  http://geco.mines.edu/guide/icc.html
- **Wiki destiné aux développeurs et utilisateurs de Gentoo**
  http://wiki.gentoo.org/wiki/GCC_optimization/fr#-O
- **Cours d'architecture de Monsieur Jalby**

ANNEXES

ANNEXE 1 : KERNELPARTIE1.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 ;
}
```

ANNEXE 2 : 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 cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}
```

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

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

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

      /* Warmup */
      printf("Valeur s = %f\n",kernel(size, (float (*)[size]) a));

      /* Stockage des résultats */
      float results[rept];
      float sum = 0.0;
      float denominateur;
      float numerateur;

      /* Répétitions */
      for (r=0; r<rept; r++){
            //printf("%d\n",r);
            uint64_t t1 = rdtsc();
            kernel(size, (float (*)[size]) a);
            uint64_t t2 = rdtsc();
            denominateur = t2-t1;
            numerateur = size*size;
            //printf("%.6f\n", denominateur/numerateur);
            results[r]=denominateur/numerateur;
            sum+=denominateur/numerateur;
      }
      /* Affichage performance */
      qsort(results, rept, sizeof(float), &cmpfunc);
      printf("moy = %.6f\n", (float)sum/(float)rept);
      printf("min = %.6f\n", results[0]);
      printf("max = %.6f\n", results[rept-1]);
      printf("med = %.6f\n", results[rept/2]);
      return 0;}
```

ANNEXE 3 : KERNELPARTIE2OPTIM1.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 ;
}
```

ANNEXE 4 : KERNELPARTIE2OPTIM2.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 ;
}
```

ANNEXE 5 : KERNELPARTIE2OPTIM3.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 ;
}
```

ANNEXE 6 : RAPPORT MAQAO DU NOYAU COMPILE EN GCC-O2

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

Partie2Optim1 - MAQAO

################################

Section 1: Function: kernel

============================



Found no debug data for this function.

With GNU or Intel compilers, please recompile with -g.

With an Intel compiler you must explicitly specify an optimization level.
```

Alternatively, try to:

- recompile with -debug noinline-debug-info (if using Intel compiler 13)

- analyze the caller function (possible inlining)

Section 1.1: Binary loops in the function named kernel

==========

Section 1.1.1: Binary loop #3

==========

Location

The loop is defined in -1:-1--1

In the binary file, the address of the loop is: 4009d0

Type of elements and instruction set

---

1 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in scalar mode (one at a time).

Vectorization

Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode).

Only 12% of vector length is used.

Matching between your loop (in the source code) and the binary loop

---

The binary loop is composed of 1 FP arithmetical operations:

- 1: addition or subtraction

The binary loop is loading 4 bytes (1 single precision FP elements).

Arithmetic intensity is 0.25 FP operations per loaded or stored byte.

Cycles and resources usage

Assuming all data fit into the L1 cache, each iteration of the binary loop takes 3.00 cycles. At this rate:

- 2% of peak computational performance is reached (0.33 out of 16.00 FLOP per cycle (0.80 GFLOPS @ 2.40GHz))

- 4% of peak load performance is reached (1.33 out of 32.00 bytes loaded per cycle (3.20 GB/s @ 2.40GHz))

Pathological cases

Performance is bounded by DATA DEPENDENCIES.

By removing most critical dependency chains, you can lower the cost of an iteration from 3.00 to 1.00 cycles (3.00x speedup).

Two propositions:

- Try another compiler or update/tune your current one:

- Remove inter-iterations dependences from your loop.

Fix as many pathological cases as you can before reading the following sections.

Bottlenecks

By removing all these bottlenecks, you can lower the cost of an iteration from 3.00 to 1.00 cycles (3.00x speedup).

All innermost loops were analyzed.