2013/2014

Architecture

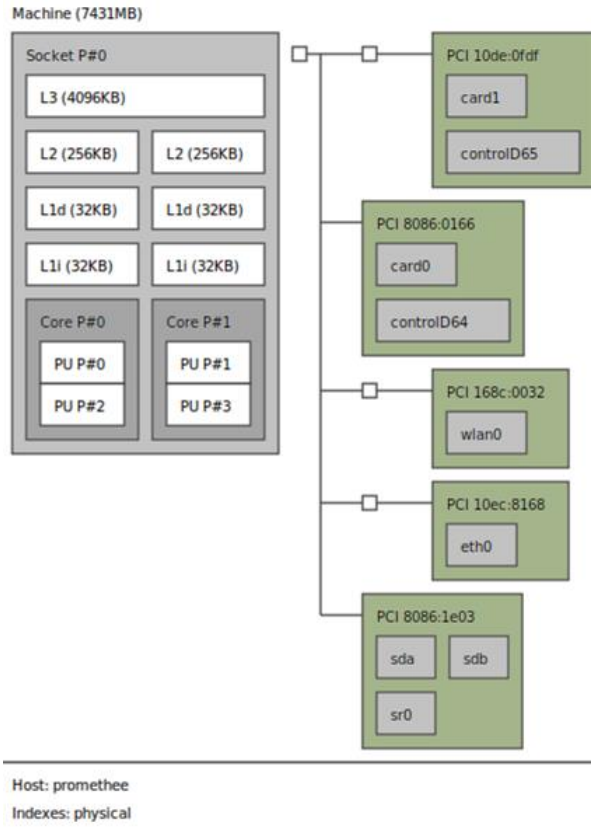
ETUDES DE CAS – SUJET 5 – CACHE L1

XAVIER NORBAL

Table des matières

I. Introduction

L'étude suivante a été faite sur une machine dont l'architecture du processeur est la suivante :



La version de GCC utilisée est **4.8.1**

La version d'ICC est **14.0.2**

La version de maqao **2.1.1**

Les performances données correspondent au nombre de cycle par itération de la boucle centrale du noyau.

II. Partie 1

A. Driver

Pour réaliser les appels à la fonction kernel, le driver suivant a été écrit :

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


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

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));
      for (r=0; r<rept; r++){
            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;
        }
        //printf("%d\n",r);
        /* 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;
}
```

Le premier paramètre indique la taille de la matrice à utiliser, le deuxième le nombre d'itérations à faire pour les mesures.

On initialise chaque cellule de la matrice avec une valeur aléatoire entre 0 et 1. Ensuite vient l'étape de warmup ou on donne un peu d'élan au processeur avant d'effectuer les vraies mesures. On récupère à chaque exécution le temps moyen d'une itération de boucle. Ensuite on trie toute ces mesures afin de récupérer le min, le max et la médiane, puis l'on fait la moyenne.

B. gcc –O2

1. Compilation

```
gcc -O2 -c kernel_orig.c -o obj/kernel_orig.o
gcc -o bin/O2orig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/O2orig 500 1000
Valeur s = 124906.406250
moy = 2.421961
min = 2.419612
max = 2.642888
med = 2.419756
```

3. 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
```

4. Déductions

Nous avons là la base de notre étude, c'est à ce résultat que nous comparerons les autres.

C. gcc –O3

1. Compilation

```
gcc -O3 -c kernel_orig.c -o obj/kernel_orig.o
gcc -o bin/O3orig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/O3orig 500 1000
Valeur s = 124906.406250
moy = 2.423102
min = 2.419636
max = 3.012084
med = 2.419744
```

3. 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
```

1. Déductions

Le code assembleur entre l'optimisation gcc –O2 et –O3 est identique, c'est pourquoi les performances sont sensiblement similaires

D. gcc –O3 –march=native

1. Compilation

```
gcc -O3 -march=native -c kernel_orig.c -o obj/kernel_orig.o
gcc -o bin/O3marchorig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/O3marchorig 500 1000
Valeur s = 124906.406250
moy = 2.435281
min = 2.420084
max = 4.931952
med = 2.420364
```

3. Assembleur

```
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
```

4. Déductions

La directive –march=native prend en compte l'rchitecture du processeur lors de la compilation, c'est pourquoi les instructions *xorps* sont devenues des *vxorps* et les *adds* des *vaddss*

E. icc –O2

1. Compilation

```
icc -O2 -c kernel_orig.c -o obj/kernel_orig.o
icc -o bin/iccO2orig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/iccO2orig 500 1000
Valeur s = 124908.843750
moy = 0.335933
min = 0.333188
max = 0.391696
med = 0.335708
```

3. Déduction

Le code asssembleur produit par icc est beaucoup plus long et complexe, on peut comprendre que c'est le plus adapté pour l'architecture présente sur la machine. C'est pourquoi la performance est aussi bonne.

F. icc –O3

1. Compilation

```
icc -O3 -c kernel_orig.c -o obj/kernel_orig.o
icc -o bin/iccO3orig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/iccO3orig 500 1000
Valeur s = 124908.843750
moy = 0.337815
min = 0.333368
max = 0.573624
med = 0.335660
```

3. Assembleur

Le code assembleur produit par icc –O3 est identique au code produit par icc –O2. C'est la même problématique que pour gcc –O3 et gcc –O2. Les performances sont donc similaires.

G. icc –O3 –xHost

1. Compilation

```
icc -O3 -xHost -c kernel_orig.c -o obj/kernel_orig.o
icc -o bin/iccO3xorig obj/kernel_orig.o driver.c
```

2. Performance

```
./bin/iccO3xorig 500 1000
Valeur s = 124908.812500
moy = 0.363908
min = 0.362084
max = 0.424740
med = 0.363528
```

3. Assembleur

L'utilisation de la directive –xHost permet l'utilisation du plus haut niveau de vectorisation supporté par le processeur. Le code assembleur s'en trouve alourdi, et ce n'est pas plus efficace qu'icc –O3

III. Partie 2

Dans cette partie, les mesures se font dans le cache L1. Un float fait 4 Octets et l'architecture du processeur présente un L1 de 32 Ko.

Soit une contenance de 1024*32/4=8192 float

Ceci donne de la place pour une matrice carrée de taille=√8192=90. Nous allons donc faire nos tests sur des matrices carrées de taille 90

Likwid-perfctr ne possédant pas d'options pour étudier le cache L1, c'est pourquoi on ne peut pas l'utiliser pour l'analyse du noyau.

A. Noyau original

1. Performance du noyau

```
./bin/O2orig 90 1000
Valeur s = 4042.028564
moy = 2.434522
min = 2.422346
max = 5.202592
med = 2.422716
```

2. Analyse

```
./bin/maqao cqa ./bin/O2orig loop=0
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 #0
=============================

Location
--------
The loop is defined in -1:-1--1
In the binary file, the address of the loop is: 4006c0

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 (0.67 GFLOPS @ 2.00GHz))
 - 4% of peak load performance is reached (1.33 out of 32.00 bytes loaded
per cycle (2.67 GB/s @ 2.00GHz))

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 1.33 cycles (2.25x 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 1.33 cycles (2.25x speedup).


All innermost loops were analyzed.
```

3. Goulet d'étranglement

D'après Maqao :

```
Performance is bounded by DATA DEPENDENCIES.
By removing most critical dependency chains, you can lower the cost of an
iteration from 3.00 to 1.33 cycles (2.25x speedup).
Two propositions:
 - Try another compiler or update/tune your current one:
 - Remove inter-iterations dependences from your loop.
```

La performance est donc limitée par les dépendances de données et peut être optimisée en retirant les dépendances inter-itérations.

De plus, on constate que la matrice est lue colonne par colonne alors qu'elle est rangée ligne par ligne, ce qui va générer des cache miss. On décide donc pour une première optimisation d'inverser les deux boucles et de lire la matrice dans le sens des lignes. Et d'une deuxième où l'on fera du loop unroll.

B. Optimisation 1

```
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 [ j ][ i ];
            }
      }
      return s ;
}
```

On se contente de lire la matrice en ligne et non en colonne.

1. Performance

```
gcc -O2 -c kernel_op1.c -o obj/kernel_op1.o
gcc -o bin/op1 obj/kernel_op1.o driver.c
./bin/op1 90 1000
Valeur s = 4042.036133
moy = 2.432027
min = 2.422346
max = 5.216173
med = 2.422346
```

2. Analyse

La performance reste la même. Ceci s'explique par le fait que

- La matrice tient dans le cache L1
- A chaque cache miss, une la ligne correspondante de la matrice est mise en cache
- A un moment toute la matrice est en cache
- Du coup il y a autant de miss dans le kernel original que dans l'optimisation

C. Optimisation 2

```
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 ;
}
```

1. Performance

```
gcc -O2 -c kernel_op2.c -o obj/kernel_op2.o
gcc -o bin/op2 obj/kernel_op2.o driver.c
./bin/op2 90 1000
Valeur s = 4042.031006
moy = 2.517109
min = 2.500124
max = 5.334938
med = 2.505062
```

2. Analyse

La performance n'est pas améliorée, et est même moins bien que la référence. En essayant de retirer les dépendances inter itération, nous avons complexifié le code, ce qui peut expliquer ce résultat.