Université Versailles-Saint-Quentin

Paris-Saclay

UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Spécialité :
Calcul Haute Performance, Simulation

Module :
Architecture et Optimisation de Code

February 21, 2025

Rapport :
Rapport TP1.

Présenté par:
- Sofiane ARHAB.

Encadré par:
- Emmanuel Oseret.

Contents

List of Figures

List of Tables

1 Description de l'architecture cible :

La suite de ce TP sera réaliser sur l'architecture cible suivante, on décris également les différentes versions des programmes utilisé :

Modèle	Coeurs	Fréquence(GHz)	L1(KiB)	L2(MiB)	L3(MiB)	Cache line (o)
AMD Ryzen 7 6800H	8	1.6 - 4.7	256	4	16	64

Table 1: Architecture cible (x86_64)

Compilateur	Version
GCC	13.1.0
OpenMP	201511
MAQAO	2.21.1
OS	Linux Mint 21.2 x86_64

Table 2: Versions programmes utilisé

2 Protocole de mesure

Afin de garantir une stabilité et une reproductibilité des résultats, on s'assure de mettre en oeuvre les points suivants

- Brancher le laptop à la prise d'alimentation.
- Fermer toutes application tierces, les autre processus indésirable et couper le réseau.
- Tuer l'interface graphique `Ctrl + ALt + F3` (tty3).
- Définir le Gouverneur du CPU en mode "performance" avec la commande `sudo cpupower frequency-set -g performance`.
- Les optimisations seront cumulatives , chaque nouvelle optimisation se base sur la précédente.
- Durant les optimisation de nature séquentielle on pineras l'exécutions de notre kernel sur un cœur unique avec la commande `taskset -c 3 ./measure [Taille] [Warmup] [Measure]`. afin d'éviter les effets de migration du processus entre les cœurs fait par l'OS.
- De même durant les optimisation parallèle utilisant OPENMP, on utiliseras `OMP_PLACES=cores OMP_PROC_BIN`

3 Kernel

Le kernel que nous optimiserons par la suite est le Kernel n°13

Kernel

```
void baseline (unsigned n ,double[n] ,const float b[n] ,const float c[n] ,
               const float d[12]){
    unsigned k , i ;
    // all elements in b are assumed positive
    for (k =0; k <12; k++){
        for (i =0; i < n ; i++) {
            if (b[i] >= 0.0 && b[i] < 1.0) {
                a[i] += exp (b[i] + d [k]) / c[i];
            } else if ( b[i] >= 1.0) {
                a[i] += (b[i] * d[k]) / c[i];
            }
        }
    }
}
```

On a à priori une complexité en $O(N^2)$, vu qu'on a deux boucles imbriqués, bien que la boucle sur k est de taille fixe (12), on a également un appel de la fonction mathématique exponentielle très coûteux, des divisions ainsi que la présence d'un branchement conditionnelle dans la boucle interne. Le cache L1 de notre machine est de taille `256 (Kib)` pour 8 coeurs, pour se ramener à un seul coeur `256/8 = 32 (Kib) = 32768 octets`. par la suite on traiteras le problème avec des tableaux de taille `n = 2000`, ce qui nous donne $n * (double) + n * (float) * 2 + 12 * (float) = 32048 octets$ on se place donc juste en dessous de la limite de notre cache L1 et viserons principalement celui-ci. La suite des résultats sera exécuté avec 100 répétitions de warmup et 100000 répétitions de mesure afin d'avoir un temps de profilage suffisant pour optimisations les optimisations les plus rapide.

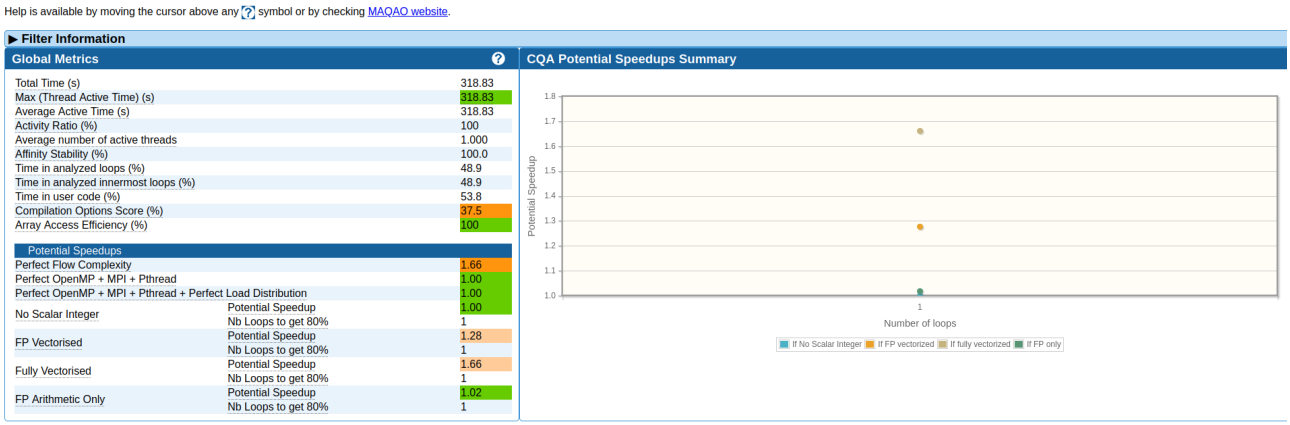

Figure 1: MAQAO - version de base

Le rapport MAQAO executé sur la version de base, nous indique déja un score un de compilation faible

Version	Temps d'exécution (s)
Baseline	318.83

Table 3: Temps d'execution par version

3.1 Optimisation 0 : Compilation

On ajoute donc les flags précédent dans notre `Makefile`, `-lm` pour linker la librairie mathématique nécessaire à l'exécution du kernel de toute manière, `-ftreevectorize` afin d'inciter le compilateur a vectoriser automatiquement si il le peux, `-funroll-loops` afin d'inciter le compilateur a dérouler les boucles si il le peux, `-fno-omit-frame-pointer` ce flag a été suggérer par MAQAO afin avoir plus de détails sur les callchains, `-march=native` pour cibler l'architecture de notre CPU (ZEN3), `-fopenmp`, pour l'utilisation d'openmp (vers les dernière optimisations), ainsi que `-Ofast` plus grand niveau d'optimisation de `GCC`, peut cependant altérer la précision numérique des résultats, une vérification sera nécessaire.

Makefile

```
OPTFLAGS=-Ofast -g -Wall -lm -ftree-vectorize -funroll-loops \
-march=native -fopenmp -fno-omit-frame-pointer
```

Le temps d'exécution relevé sur MAQAO est : `338.37 (s)`

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56

Table 4: Temps d'execution par version

3.2 Optimisation 1 : Invariants et Restrict

On remarque que la variable `d[k]` est accéder à chaque iteration de la boucle interne, alors que celle-ci ne dépends que de `k`, on la sort donc de la boucle interne pour éviter de payer l'accès à la mémoire à chaque accès, on déclare également nos tableaux comme `restrict` pour s'assurer qu'il n'existe qu'une seule copie de ces tableaux en mémoire pour éviter les effets d'aliasing.

Kernel

```
const float d_k = d[k] ;
```

C'est une optimisation très basique mais trés efficace, en vérifiant le rapport de MAQAO a ce sujet , notre temps d'exécution passe à `162.04 (s)`

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89

Table 5: Temps d'execution par version

3.3 Optimisation 2 : Fast-expo

MAQAO nous indique que la majorité du temps d'exécution du pragramme est passé dans des appels de fonction à la `libm GlibC` ce qui correspond à l'exponentielle de la GlibC, nous allons donc remplacer cet appel coûteux par une approximation du calcul de l'exponentielle, nous avons tester l'approximation de Schraudolph ainsi que l'approximation de Taylor :

Kernel

```c
// Schraudolph
static inline float fast_exp2(float x) {
    union { float f; int i; } v;
    v.i = (int)(12102203.0f * x + 1065353216.0f);
    return v.f;
}

// Taylor approximation
static inline float fast_exp(float x) {
    return 1.0f + x * (1.0f + x * (0.5f + x * (1.0f / 6.0f)));
}
```

Les deux approximation ont donné un résultats assez similaire en terme de temps d'exécution, nous avons garder la meilleure (Schraudolph)

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89
OPT2 (Fast-expo Shraudolph)	51.11
OPT2 (Fast-expo Taylor)	50.96

Table 6: Temps d'execution par version

3.4 Optimisation 3 :Changement de boucle et d'invariants

L'extraction de l'invariant précédent nous faisait économiser `12*n` accès mémoire, en intervertissant les boucles on peut dorénavant extraire 3 invariants `c[i]` `b[i]` et `a[i]` qu'on remplace par une variable local `sum` qu'on assigneras une seule fois à `a[i]` une fois sortie de la boucle interne,

Kernel

```c
void kernel(unsigned n, double* restrict a, const float* restrict b, \
const float* restrict c ,const float* restrict d) {
   unsigned k, i ;
for(i=0 ; i<n ; i++){
      const float b_i = b[i] ;
      const float c_i = c[i] ;
      double sum = 0.0 ;
      for(k=0; k<12; k++){
         if(b_i >= 0.0 && b_i < 1.0){
           sum += fast_exp2(b_i + d[k]) / c_i;
         } else if (b_i >= 1.0) {
            sum += (b_i * d[k]) / c_i ;
         }
      }
      a[i] = sum ;
   }
```

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89
OPT2 (Fast-expo Shraudolph)	51.11
OPT2 (Fast-expo Taylor)	50.96
OPT3 (Loop-change)	19.50

Table 7: Temps d'exécution par version

3.5 Optimisation 4 : Flow-Complexity

MAQAO nous indique un score de complexité assez faible, cela est dû à la présence de branchement conditionnelle `if-else` dans la boucle interne, cela empèche également le compilateur de vectoriser, nous allons donc essayer de se débarasser de ces branchement en faisant du masking

Kernel

```
void kernel(unsigned n, double* restrict a, const float* restrict b, \
const float* restrict c ,const float* restrict d) {
   unsigned k, i ;
for (i = 0; i < n; i++) {
        const float b_i = b[i];
        const float c_i = c[i];
        double sum = 0.0;
        const float mask1 = (b_i >= 0.0f && b_i < 1.0f) ? 1.0f : 0.0f;
        const float mask2 = (b_i >= 1.0f) ? 1.0f : 0.0f;

        for (k = 0; k < 12; k++) {
            sum += mask1 * (fast_exp2(b_i + d[k]) / c_i) +
                   mask2 * ((b_i * d[k]) / c_i);
        }
        a[i] = sum;
    }
}
```

On obtient notre meilleure taux de vectorization juesu'à présent à `99.56%`.

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89
OPT2 (Fast-expo Shraudolph)	51.11
OPT2 (Fast-expo Taylor)	50.96
OPT3 (Loop-change)	19.50
OPT4 (Complexity-Reduction)	15.56

Table 8: Temps d'exécution par version

3.6 Optimisation 5 : OPEN-MP

Nous allons maintenant introduire du parallélisme vu qu'à priori notre double boucle ne présente aucun loop-dépendance et que le calul peut se paralléliser, nous allons donc simplement introduire le pragma `#pragma omp parallel for`, et définir `OMP_NUM_THREADS=4 , OMP_NUM_THREADS=8` , avec l'introduction d'openMP, on n'utiliseras plus `taskset -c 3` pour pinner l'exècution du code sur un coeur particulier vu qu'on fait appel à plusieurs coeurs désormais, mais à la place on va définir les variables d'environnement suivantes comme préconisé par MAQAO `OMP_PLACES=cores OMP_PROC_BIND=close`

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89
OPT2 (Fast-expo Shraudolph)	51.11
OPT2 (Fast-expo Taylor)	50.96
OPT3 (Loop-change)	19.50
OPT4 (Complexity-Reduction)	15.56
OPT5 (OpenMp 4 Threads)	7.58
OPT5 (OpenMp 8 Threads)	5.33

Table 9: Temps d'exécution par version

3.7 Optimisation 5 : Inverse division

Pour finir nous allons faire remplacer les deux division par des multiplcation en calculant une seule fois l'inverse `c_i_verse` dans la boucle externe.

Version	Temps d'exécution (s)
Baseline	318.83
OPT0 (Compilation)	317.56
OPT1 (Invariant)	150.89
OPT2 (Fast-expo Shraudolph)	51.11
OPT2 (Fast-expo Taylor)	50.96
OPT3 (Loop-change)	19.50
OPT4 (Complexity-Reduction)	15.56
OPT5 (OpenMp 4 Threads)	7.58
OPT5 (OpenMp 8 Threads)	5.33
OPT6 (Inverse 4 Threads)	7.46
OPT6 (Inverse 8 Threads)	5.30

Table 10: Temps d'exécution par version

Comme les deux dernières runs d'optimisation ne prennent pas assez de temps à s'exécuter pour que MAQAO considèrent que le temps de profilage soit suffisant, nous les avons re-runs avec un nombre de répétitions de mesures de, nous n'avons pas pu re-runs toutes les versions avec ce nouveau paramètre car cela prendrait beaucoup trop de temps surtout pour la version de base, nous incluons donc les résultats des deux dernières run d'optim avec un temps de profilage suffisant.

Version	Temps d'exécution (s)
OPT5 (OpenMp 4 Threads)	38.22
OPT5 (OpenMp 8 Threads)	27.10
OPT6 (Inverse 4 Threads)	37.45
OPT6 (Inverse 8 Threads)	5.30

Table 11: Temps d'exécution par version avec 500000 mesure de répétitions

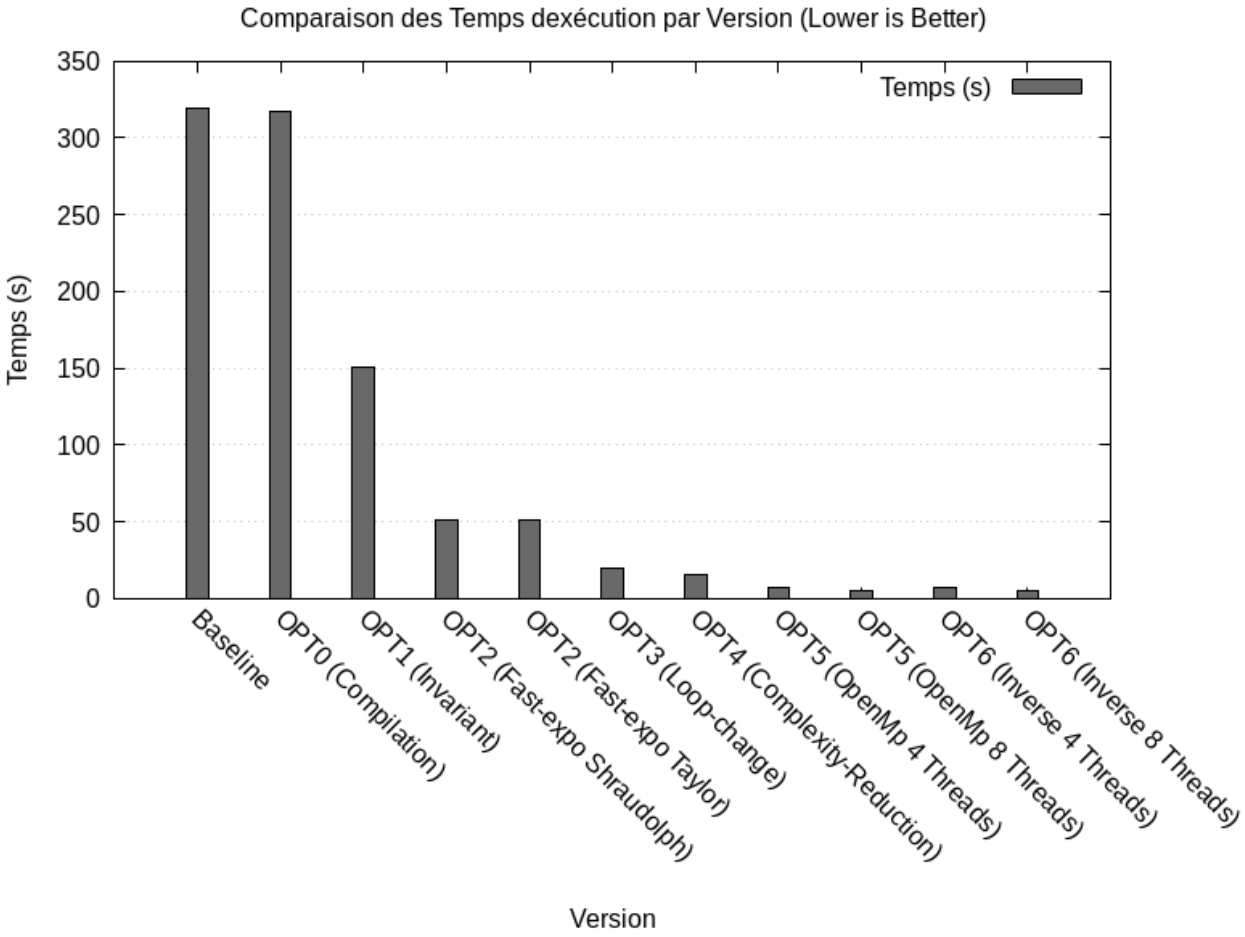

Figure 2: Temps d'exécution des différente version N =2000 repw =100 repm =100000

L'intégralité de ce travail est disponible dans ce dépot Github.