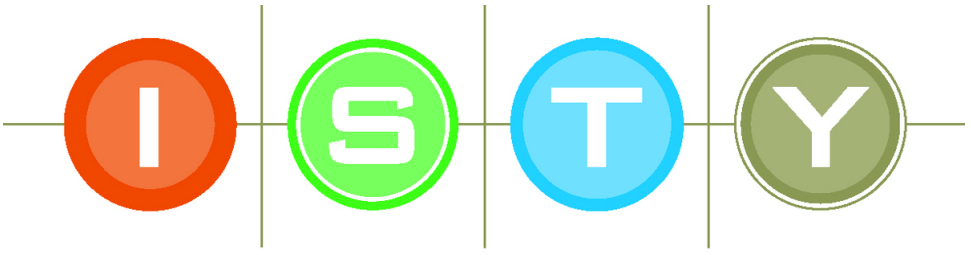
Analyse de performance et optimisation de code

AYOUB Pierre – BONNAFOUS Camille – FLAMANT Océane

17 mars 2019



INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

Résumé

La simulation numérique est un procédé informatique visant à modéliser un phénomène par ordinateur, s'agissant le plus souvent d'un phénomène physique. Cette modélisation prend forme par des systèmes d'équations décrivant l'état du système physique représenté à chaque instant. De nombreux domaines scientifiques convergent vers la simulation informatique, tel que certaines branches de la physique, de l'analyse et de l'optimisation mathématique, ou encore le calcul haute performance en informatique. Enfin, la simulation trouve naturellement de nombreuses applications concernant des sujets variés, tel que la simulation du climat et des évènements météorologiques, la simulation d'essais nucléaires, de l'effet d'un médicament sur un corps, ou encore des astres et de l'univers. Ce rapport s'articulera donc autour de l'analyse et de l'optimisation d'un code de calcul, coeur des simulations numériques présentés ci-dessus.

Table des matières

1 Introduction

Le projet que nous vous présentons aujourd'hui consiste à analyser puis, grâce à nos mesures, optimiser un code de calcul, appelé kernel. Les mesures doivent s'effectuer à l'aide de l'instruction *x86 RDTSC*, et des deux outils d'analyse de performance suivant : *MAQAO* et *LIKWID*. *RDTSC* nous permet de mesurer le nombre de cycle entre deux instants, *MAQAO* rends possible l'exécution d'analyses statiques (*CQA*) et dynamiques (*LPROF*) d'un binaire, présentées par un rapport haut niveau à l'aide de *ONE-VIEW*, enfin *LIKWID* permet d'obtenir un grand nombre de métriques très précises concernant, notamment, l'usage de la mémoire.

Afin d'étudier les différents niveaux de la hiérarchie mémoire, chaque membre du groupe analysera un niveau qu'il se verra assigné. Ci-dessous la liste des assignatons :

Pierre Cache L1 :
- Intel Core i7-6600U
- 32 kiB L1i, 32 kiB L1d
- 256 kiB L2

Océne Cache L2 :
- Cache L1 : 32K
- Cache L2 : 256K
- Fréquence 2,40GHz

Camille RAM :
- TO DO

Le déroulement du projet s'est effectué en plusieurs étapes distinctes :

Analyse du code Cette phase consiste à analyser le programme d'un point de vue d'architecture informatique. Il convient d'étudier les choix mis en œuvres afin d'implémenter le ou les calculs nécessaires.

Protocole expérimental Une fois l'analyse effectuée, nous pouvons en déduire le moyen le plus adapté afin de mesurer les performances de notre implémentation. Nous allons donc mettre en avant les critères théoriques à atteindre dans nos mesures, puis nous exposerons la manière dont nous avons mis ceci en pratique.

Optimisations et mesures Grâce au protocole mis en place, nous pouvons quantifier la performance du programme. De ce fait, nous serons en mesure d'expérimenter différentes techniques d'optimisation sur le programme et d'en calculer l'accélération.

2 Analyse du code

Présentons notre kernel par son prototype, que nous observons sur le Listing 1 Nous voyons que 3 variables manipulées :

- n correspond à la taille de nos tableaux.
- a est un tableau de *float* à deux dimensions dont la taille en fonction de n : $4n^2$ Bytes.
- b est un tableau de *double* à une dimension dont la taille en fonction de n : $8n$ Bytes.

```
1 void baseline(unsigned n, float a[n][n], double b[n])
```

Listing 1 – Prototype du kernel non-optimisé

Nous sommes face à un code de calcul très simple en apparence, illustré dans le Listing 2 : deux boucles imbriquées, un branchement, un calcul mêlant multiplication et exponentiel. Plusieurs éléments remarquables qui risquent de poser problème au niveau de la rapidité d'exécution apparaissent alors : les boucles impliquent qu'il faut prêter attention au sens de parcours des tableaux, le branchement nous laisse penser qu'il faudrait essayer de le supprimer, enfin l'exponentiel et la multiplication sont des opérations lourdes.

```
1 for (j = 0; j < n; j++) {
2     for (i = 0; i < n; i++) {
3         if (j == 0)
4             b[i] = 1.0;
5         b[i] *= exp(a[i][j]);
6     }
7 }
```

Listing 2 – Kernel non-optimisé

3 Protocole expérimental

La mise en place d'un protocole expérimental de mesure est une étape nécessaire et cruciale dans tout processus d'optimisation de code. D'une part, le but de ce protocole est de mettre en lumière les points chauds du programme, c'est-à-dire les parties du code qui ralentissent considérablement l'exécution des calculs : ces points chauds seront les cibles de nos optimisations. D'autre part, après chaque tentative d'optimisation, le protocole doit nous permettre de mesurer l'impact de cette dernière, qu'il soit positif ou négatif, et enfin de le quantifier.

3.1 Théorie

Lors de nos expériences, de nombreux paramètres peuvent être sujets à des variables aléatoires ou à l'erreur de mesure, ainsi un résultat peut être biaiser. Afin d'éviter cela, il est impératif d'utiliser une valeur représentative de nos différentes mesures : une valeur moyenne ou une valeur médiane. La valeur médiane semble être un meilleur candidat contrairement à la moyenne, car cette dernière peut-être fortement modifié par une valeure extrême qui n'as pas lieu d'être. C'est donc la valeur médiane que nous prendrons des réslutats de nos mesures successives issues des méta-répétitions.

Pour que chaque membre de l'équipe puisse travailler sur son niveau de mémoire cache, il nous faut trouver la taille des données d'entrée à utiliser. Selon le protoype présenté dans le Listing 1, la taille totale de nos données d'entrées est de $4n^2 + 8n$, avec n la taille rentrée en paramètre du programme.

3.1.1 Cache L1

Ci-dessous les paramètres des mesures :

(1) Taille pour tenir dans L1 : $n = 88$. En effet, $(n^2 * 4) + (n * 8) = 31680B$, sachant que la taille du cache est de $32kiB = 2^{15} = 32768B$, et que si l'on prends $n = 90$, on obtient $33120B$, on a bien : $n = 88 < L1 < n = 90$ avec une bonne marge de sécurité.

(2) Nombre de répétition du warm-up : 1000. Ce nombre est suffisant pour avoir ensuite des mesures stables dans tout nos tests car les caches sont remplis avec nos tableaux, choisi par observation.

(3) Nombre de répétition des mesures : on choisi un nombre qui nous permet d'avoir à peu près 3 secondes par méta-répétition. Ce nombre vari en fonction de la taille de notre tableau. Avoir quelques secondes de mesures permet d'avoir une faible marge d'erreur de mesures des cycles avec un RDTSC. Par exemple, pour une taille de 89, on peut choisir 1000000.

3.1.2 Cache L2

Pour que les deux tableaux entrent entièrement dans le cache L2, il faut que la formule respecte les contraintes suivantes :

— La taille totale doit être plus grande que la taille du cache L1 (1). Pour plus sécurité il a été décidé que la taille totale devait être au moins trois fois plus grande que celle du L1.

— L2 partage sa mémoire pour stocker à la fois les instructions, les données et ce qui tourne en background, on ne peut donc en utiliser approximativement que 90% (2).

Ces deux contraintes peuvent être transformée sous forme d'inéquation :

— (1) : $3 * TL1 < 4n * n + 8n$,

— (2) : $4n * n + 8n \leq 0,9 * \text{TL2}$

Après la résolution de ces équations on obtient $n = 156$ comme minimum et $n = 242$ comme maximum.

3.1.3 RAM

TODO

3.1.4 Analyse de sensibilité

Une fois que l'on connait la taille des données à fournie en entrée, il faut effectuer une analyse de sensibilité pour les autres paramètres.

Nombre de méta-répétition des mesures Il nous est donné à 31. C'est le nombre donné dans la consigne, qui est suffisant pour avoir un nombre de mesure significatives.

Le nombre de warmup Il doit se situer entre 1 et 1000. Pour le déterminer, il doit être le seul paramètre que l'ont fait varier. On fait plusieurs exécutions et, avec les valeurs obtenues, on fait une courbe pour voir à partir de quelle valeur cela devient stable. Il faut aussi vérifier que $\frac{\text{médiane}-\text{minimum}}{\text{minimum}}$ est inférieur à 5%.

Le nombre de répétitions On le trouve de la même manière que le nombre de warmup.

3.2 Pratique

Lors de nos première tentatives pour trouver les paramètres, nous avons remarqué que ce qui prenait le plus de temps dans notre noyau de calcul était l'exponentiel. Afin de pouvoir vérifier si nos paramètres sont corrects, nous avons donc modifié le fichier *kernel.c* pour que ce soit le temps de récupération des données qui soit le plus grand. Cette version n'est utilisé que pour tester la véracité des paramètres trouvés dans la section précédente, et nous pour évaluer les performances des optimisations ou des compilateurs.

3.2.1 Cache L1

On peut observer une nette différence de performance entre un $n = 88$ et un $n = 90$, qui se traduit par le fait de tenir ou de ne pas tenir en cache *L1*. On assigne le programme de calcul au coeur n°2 de la machine, sur son numéro de thread physique, dans le but de limiter les changements de contexte et de flush de la mémoire cache. Enfin, les tests sont effectué en rescue mode, sans interface graphique, avec le minimum de tâches tournant en arrière plan.

Tout les mesures effectuées dans le cache L1 sont automatiser par un script (*bench.sh*). Ce script permet d'aisément exécuter l'ensemble des tests dans un environement idéal, ainsi que d'assurer la reproductibilité du protocole expérimental, critère important d'une méthode scientifique.

3.2.2 Cache L2

Pour vérifier le calcul théorique de la taille des données j'ai utilisé likwid-perftcr afin de voir si les données transitaient bien par le cache L2. Après avoir compilé avec gcc uniquement j'ai exécuté l'exécutable avec likwid et voici les résultats obtenus :

n	Data Volume (GByte)
100	4,24
150	14,06
220	37,9
235	34,4

On observe que ces résultats sont en corrélation avec les résultats théoriques en dessous de 156 pratiquement aucune donnée ne passe par le cache L2 et quand on se rapproche de 242 une partie des données ne semble plus passer dans L2 je suppose donc que ces données vont directement dans le cache L3. Au vu de ces informations, j'ai choisi de prendre 220 comme taille de données.

Voici, ci-contre le graphique obtenu pour trouver le bon nombre de warmup. On peut observer que le nombre de cycle semble se stabiliser au alentour de 100 warmup. Pour plus de sécurité j'ai choisit 150 pour le nombre de warmup.

J'ai ensuite vérifié avec le calcul de l'exponetiel et on obtient bien le même résultat.

Pour trouver le bon nombre de répétition j'ai uniquement fait les test avec l'exponentiel. Comme vous pouvez le voir, on peut remarquer que l'ensemble est stable, les variations sont minimes. J'ai chosit comme nombre de répétition 1200.

3.2.3 RAM

TODO

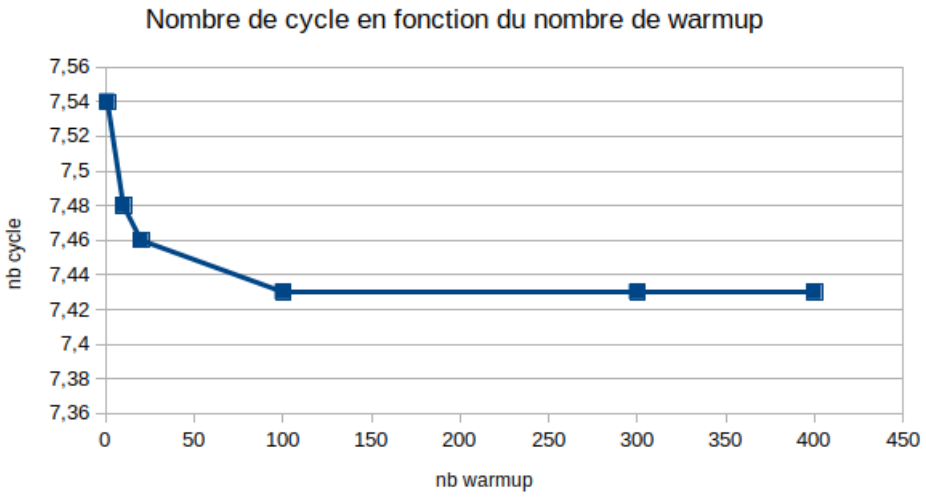


FIGURE 1 – Sans l'exponentiel

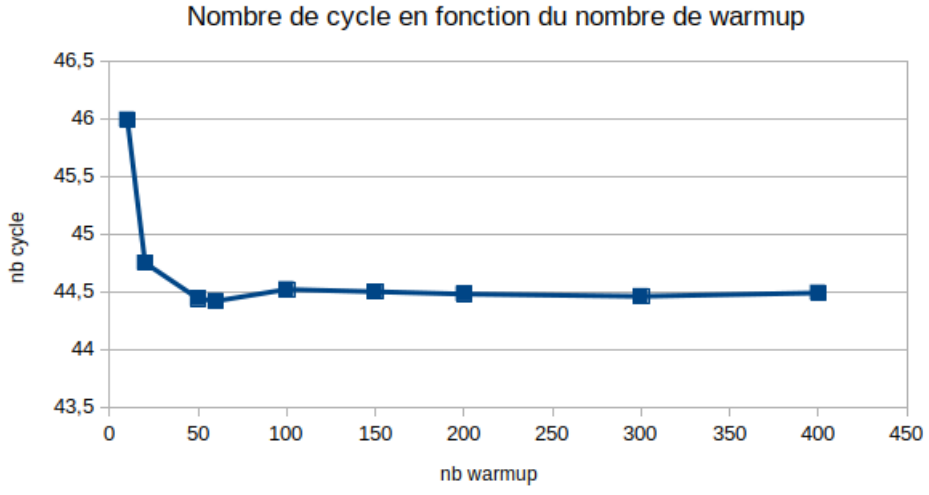


FIGURE 2 – Avec l'exponentiel

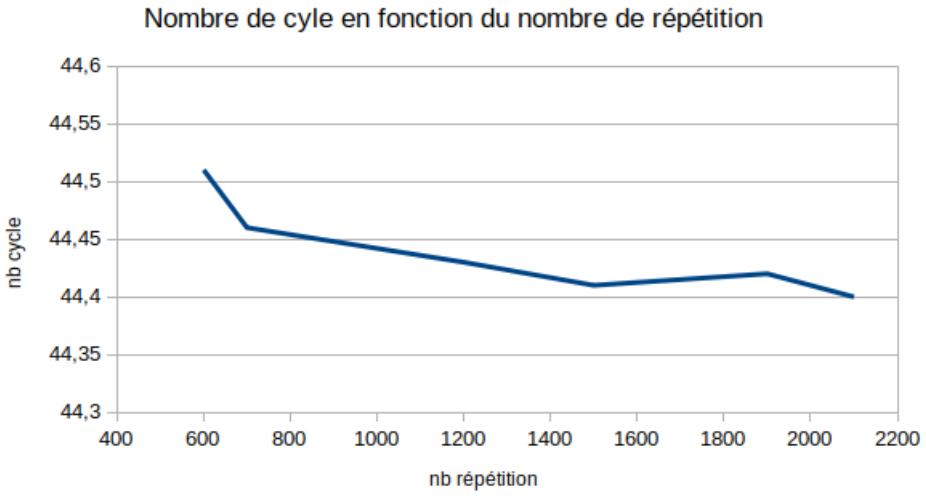

Nombre de cyle en fonction du nombre de répétition
44,6
44,55
44,5
44,45
44,4
44,35
44,3
nb cycle
400
600
800
1000
1200
1400
1600
1800
2000
2200
nb répétition

FIGURE 3

4 Optimisations et mesures

Dans cette section, nous présentons les résultats des mesures des différentes tentatives d'optimisation du code. La phase 1 correspond, pour résumer, à identifier les points chauds et tester différentes configuration de compilation. La phase 2 correspond, notamment, à une optimisation active du code en y apportant des modifications.

4.1 Phase 1

4.1.1 Cache L1

Pour cette première phase de test sur un jeu de donneé dans le niveau de cache L1, nous avons testé 3 compilateurs (« gcc », « icc », « clang ») avec différents jeu de flags de compilation. Nous précisons que l'intégralité de nos résultats sont disponibles sous formes brut dans les fichiers/répertoires suivants : « compil.txt », « likwid_{ref,opt} », « maqao_{ref,opt} ».

En plus des flags qui sont donnés dans la consigne, nous avons également testé le flag « -Ofast » qui permet d'activer des optimisation mathématiques qui ne respectent pas les standards en vigueur. Un programme qui ne requiert pas une stabilité numérique très précise obtiendra des gains considérable avec cette option, cependant, cela peut-être dangereux de l'activer sans possibilité de vérifier les résultats des calculs du kernel. Nous avons ici pris le pari de l'activer.

Nous avons aussi testés d'autres options ciblées par ce qui consitue notre code : fonctions inline, optimisations sur les boucles, sur les fonctions mathématiques ou encore sur les branchements. La liste des flags ci-dessous n'aura pas apporté de gain, ou pire, aura provoqué une accélération négatif par rapport à « -Ofast -march=native » : « -faggressive-loop-optimizations », « -fbranch-probabilities », « -fdelayed-branch », « -fexpensive-optimizations », « -finline-functions », « -floop-block », « -floop-interchange », « -floop-unroll-and-jam », « -funsafe-math-optimizations ». Cependant, le flag « -funroll-all-loops », permettant de forcer l'unrolling des boucles, nous aura octroyer un léger gain systématique.

Dans la Table 1 est présenté la liste des résultats sur les flags obligatoires et les flags apportant un gain (les flags inutiles ou ralentissant n'ont pas été inclu pour des soucis de visibilité). Nous pouvons ainsi voir que c'est « gcc », couplé à certaines options, qui est le plus rapide face à « clang » et « icc ». Nous notons tout de même l'efficacité redoutable de la génération de code spécialiement pour l'architecture hôte (« -march=native »), permettant d'utiliser les instructions x86 les plus récentes, et des optimisations mathématiques aggressives (« -Ofast »).

Nous avons ensuite utiliser les outils *MAQAO* et *LIKWID* pour expliquer les différences de performances entre deux versions du code. Après nos tests avec notre script permettant de détecter les flags permettant d'avoir le meilleur speed-up, nous allons étudier les différences de performances entres la version de référence « gcc

Compiler	**Flags**	**Time (s)**
gcc	-O2	22.90
gcc	-O3	22.58
gcc	-Os	47.99
gcc	-O3 -march=native	22.77
gcc	-Ofast	9.59
gcc	-Ofast -march=native	4.64
gcc	-Ofast -march=native -funroll-all-loops	4.42
clang	-O0	30.38
clang	-O2	23.18
clang	-O3	26.04
clang	-Os	23.11
clang	-O3 -march=native	26.21
clang	-Ofast	18.18
clang	-Ofast -march=native	25.97
icc	-O0	23.05
icc	-O2	16.25
icc	-O3	16.27
icc	-Os	15.96
icc	-O3 -xHost	17.08
icc	-Ofast	16.25
icc	-Ofast -xHost	16.41
icc	-Ofast -xHost -funroll-loops -unroll-aggressive	16.39

TABLE 1 – Benchmarks des compilateurs et flags

-O2 » et la version la plus rapide, « gcc -Ofast -march=native -funroll-all-loops ».

Procédons tout d'abord à une analyse rapide avec *LIKWID*, les résultats étants présentés dans la Figure 4. Nous pouvons expliquer la différence de performance par les métriques suivantes concernant la mémoire : on observe que la version optmisé à fait un nombre significativement moins important que la version de référence d'éviction de données du cache L1 (2051211 vs. 16140154), ainsi qu'un ratio de miss bien plus faible dans le cache L2 (0.0001 vs. 0.0259).

Enfin, passons à l'étude avec MAQAO. Sur le page « Global » présenté par la Figure 5, nous pouvons déjà avoir une très bonne idée des différences entre les deux binaires, justifiant d'une telle accélération (acc $= \frac{212}{41} = 5.2$). Premièrement, nous observons que sur le binaire optimisé, nous passons deux fois plus de temps dans la boucle que dans la version de référence : cela signifique que les fonctions mathématiques (multiplication, mais surtout l'exponentiel) ont été considérablement optimisées. Ensuite, nous voyons que la version de référence présente deux chemins (Flow Complexity) dans la boucle, tandis que la version optimisée ne présente qu'un chemin d'exécution possible. Nous notons aussi que l'efficacité d'accès aux données (Array Access Efficiency) à été augmenté de 20% dans la version optimisée, surement par modifications des boucles imbriquées. Enfin, nous pouvons imaginer une tentative de vectorisation de la part du compilateur pour la version optimisée.

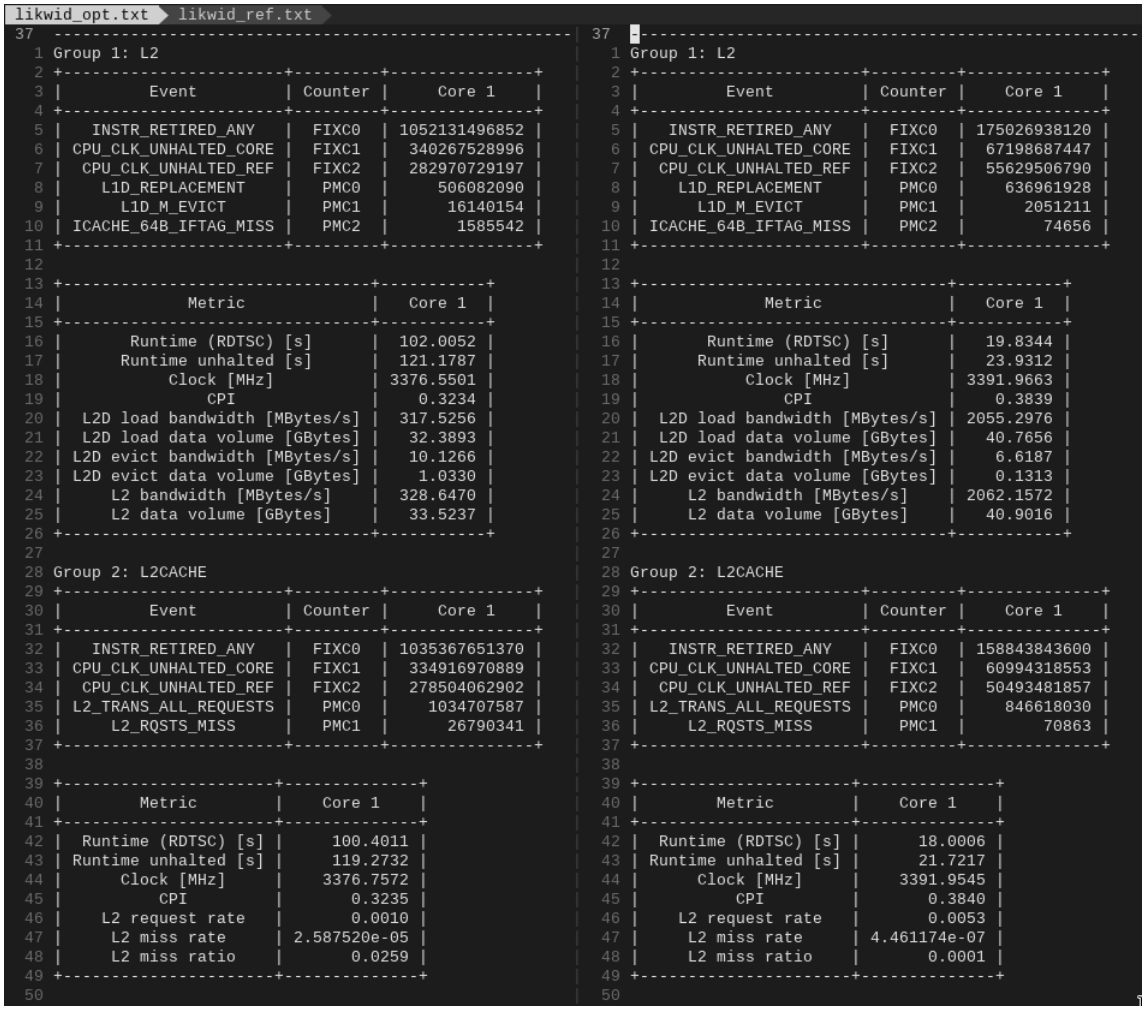


FIGURE 4 – A gauche, la version de référence. A droite, la version avec les flags d'optimisation.

Pour finir avec l'analyse de *MAQAO*, sur la Figure 6, nous pouvons observer ce qu'à concrètement fait l'optmisation. La fonction exponentiel, qui prenait 14% du temps, à été remplacé par une version optimisée « fini », ne prenant plus que 1.46% du temps. Nous pouvons voir que le linkage de la bibliothèque mathémtique (*libm*) à été remplacé par sa version vectorisé (*libmvec*). Enfin, nous observons que notre unique boucle à bien été déroulée car nous trouvons l'ajout d'une *tail loop*.

4.1.2 Cache L2

Numéro	Commande	Résultat
1	gcc -O2	? ?
2	gcc -O3	? ?
3	gcc -O3 -march=native	? ?
4	icc -O2	? ?
5	icc -O3	? ?
6	icc -O3 -xHost	? ?

4.1.3 RAM

TODO

4.2 Phase 2

TODO

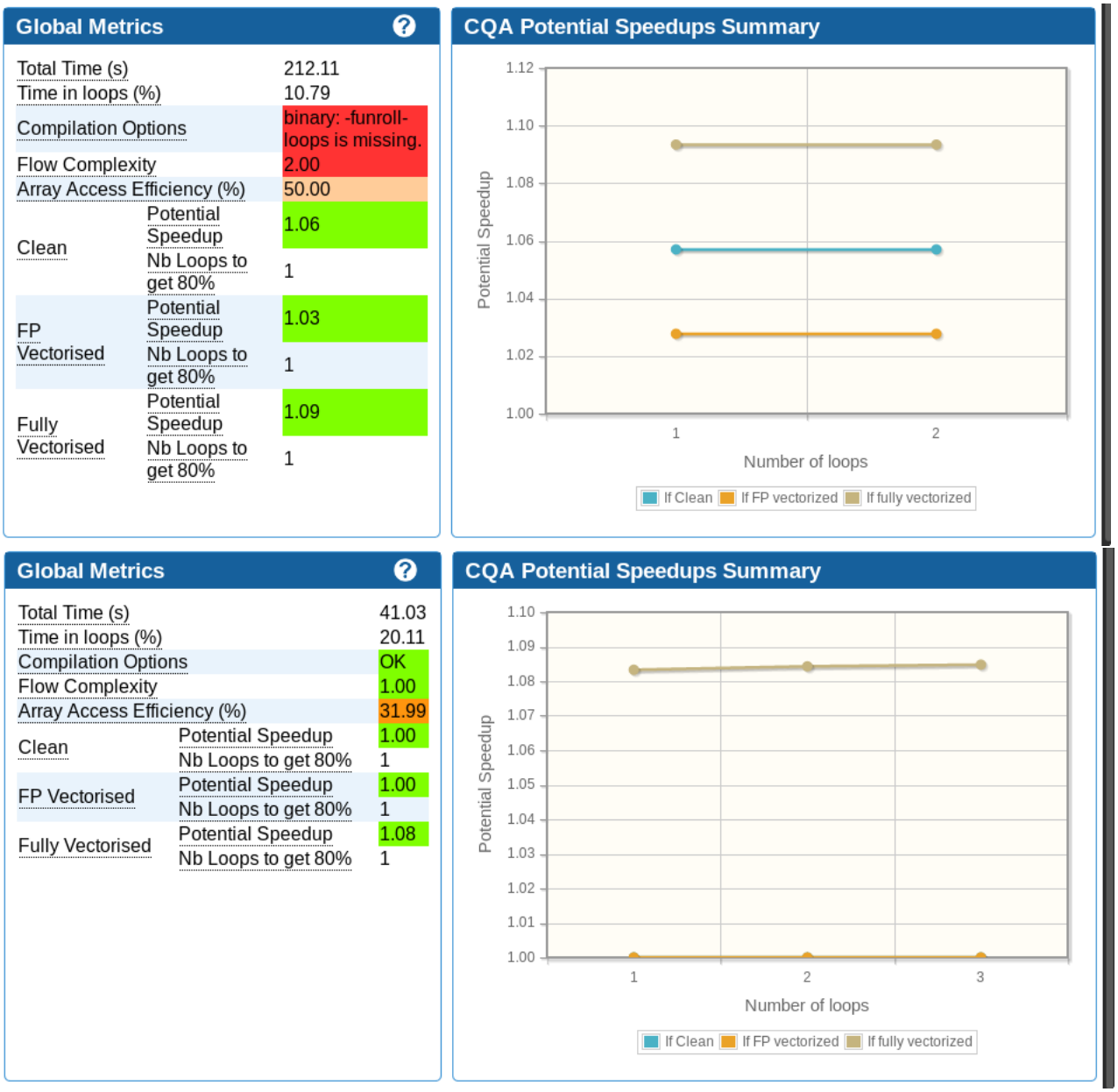


FIGURE 5 – Au dessus, la version de référence. En dessous, la version avec les flags d'optimisation.

Functions and Loops

▶ **Filters**

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ f64xsubf128	libm-2.28.so	63.65	135.01	1	0.00
○ exp	libm-2.28.so	14.61	30.99	1	0.00
▼ baseline	baseline	10.88	23.07	1	0.00
▼ Loop 7 - kernel.c:13-19 - baseline		10.79	22.89		
○ Loop 6 - kernel.c:16-19 - baseline		10.79	22.89		

Functions and Loops

▶ **Filters**

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ _ZGVeN16vvv_sincosf	libmvec-2.28.so	74.29	30.48	1	0.00
▼ baseline	baseline	24.08	9.88	1	0.00
▼ Loop 8 - - baseline		20.06	8.23		
▼ Loop 6 - - baseline		19.84	8.14		
○ Loop 7 - - baseline		19.84	8.14		
○ Loop 9 - - baseline		0.22	0.09		
○ _ZGVdN4v___exp_finite	baseline	1.46	0.6	1	0.00

FIGURE 6 – Au dessus, la version de référence. En dessous, la version avec les flags d'optimisation.

5 Conclusion

TODO

Acronymes