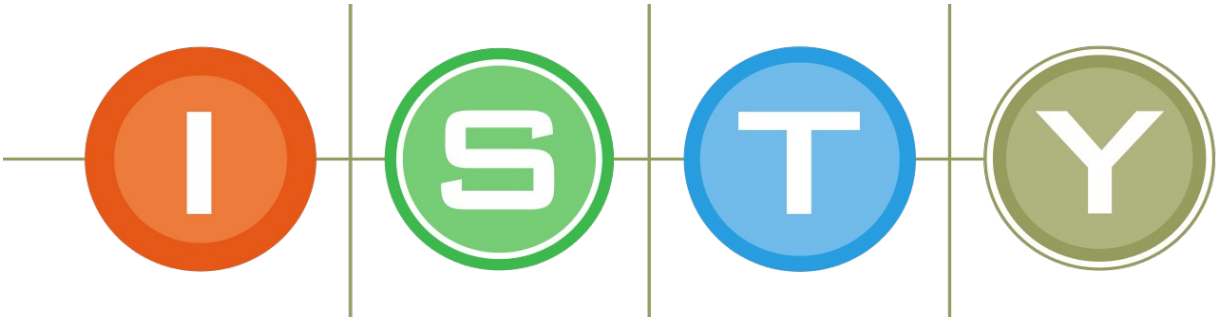


Architecture des Ordinateurs Avancée

Phase I – Sujet IV

Loïs **JULLIEN**
Clément **LEFEVRE**
Étienne **SAUVÉE**
Pierre **VERDURE**

Encadré par : M. **OZERET** et M. **LEBRAS**

Table des matières

Introduction

Ce rapport traite de la première phase du projet d'Architecture des Ordinateurs Avancées (AOA), unité d'enseignement du second semestre de IATIC4.

Pour ce projet, nous devions choisir l'un des sujets parmi les 12 proposés, nous avons donc choisi le sujet n°4.

Sujet 4

Étudier et optimiser la fonction C suivante:

```
void baseline (int n, float a[n], float b[n],
               float x) {
  int i;

  for (i=0; i<n; i++) {
    if ((i < n/2) && (a[i] > x))
      b[i] = a[i];
    else if (i < n/2)
      b[i] = x;
    else
      b[i] = a[i] + x;
  }

  for (i=0; i<n; i++) {
    if (b[i] < 0.0) b[i] = 0.0;
  }
}
```

Compilateur et options de référence: `gcc -O2`.

Illustration 1: Sujet du projet

Dans un premier temps, nous traiterons donc des travaux que nous avons effectués en commun, à savoir les procédures à mettre en place afin de réaliser nos mesures.
Ensuite, nous détaillerons les travaux effectués sur les différents niveaux de caches.

1 Travaux communs

Cette partie traitera principalement des travaux effectués en commun ainsi que des méthodes mises en œuvre afin de réaliser les mesures décrites dans les chapitres suivants.

1.1 Adaptation des fichiers

Initialement, une archive contenant plusieurs fichiers utiles à la simulation nous a été fournie. Cette archive contenait les fichiers suivants :

- kernel.c, contenant la fonction principale, que l'on cherche à optimiser.
- driver.c, contenant des fonctions relatives la simulation (initialisation des tableaux, déroulement de la simulation, avec méta-répétitions, warmups, etc.)

1.1.1 kernel.c

La modification de kernel.c, s'est avérée aisée, puisque nous avons simplement remplacé son contenu par la fonction baseline correspondant à notre sujet.

```
void baseline (int n , float a[n], float b[n], float x) {
   int i;
   for (i = 0; i < n; i++) {
      if ((i < n/2) && (a[i] > x))
         b[i] = a[i];
      else if (i < n/2)
         b[i] = x;
      else
         b[i] = a[i] + x;
   }

   for (i = 0; i < n; i++) {
      if (b[i] < 0.0)
         b[i] = 0.0;
   }
}
```

Fonction baseline

Par la suite, ce fichier contiendra également les versions optimisées de la fonction baseline.

1.1.2 driver.c

Nous avons dû apporter quelques modifications au fichier driver.c. En effet, la fonction baseline de notre sujet manipule **deux tableaux unidimensionnels** (a et b) de n **flottants** (n étant passé en paramètre du programme), ainsi qu'**une variable** (x), elle aussi donnée en entrée du programme.
Les **fonctions de manipulations de tableaux** ont donc été **réécrites** en conséquence, comme par exemple ici, pour la fonction init_array :

```
static void init_array (int n, float a[n]) {
   int i;
   for (i=0; i<n; i++)
      a[i] = (float) rand() / RAND_MAX;
}
```

Fonction init_array

Suivons le déroulement de la fonction main :

1. Dans un premier temps, on **récupère les paramètres saisis** en entrée du programme :

```
   /* get command line arguments */
   int size = atoi (argv[1]); /* matrix size */
   int repw = atoi (argv[2]); /* repetition number */
   int repm = atoi (argv[3]); /* repetition number */
   float x = atof(argv[4]); /* x value */
```

Les paramètres saisis sont convertis dans leur type respectifs, via les fonction atoi (ASCII to integer) et atof (ASCII to float).

2. On itère ensuite sur le nombre de méta-répétitions (défini à 31 par défaut), et pour chacune d'elles, on effectue les actions suivantes :

⇒ **Allocation de la mémoire** aux deux tableaux de taille size flottants, via la fonction malloc.

```
      /* allocate arrays */
      float *a = malloc (size * sizeof *a);
      float *b = malloc (size * sizeof *b);
```

⇒ **Initialisations des tableaux**, via la fonction init_array présentée précédemment :

```
    /* init arrays */
     srand(0);
     init_array (size, a);
     init_array (size, b);
```

Initialiser la fonction rand via srand(0) permet d'utiliser des valeurs pseudo-aléatoires, mais reproductibles.
Les deux tableaux sont donc remplis de ces valeurs pseudo-aléatoires.

⇒ **Phase de warmup**

```
     /* warmup (repw repetitions in first meta, 1 repet in next metas) */
     if (m == 0) {
        for (i=0; i<repw; i++)
           baseline (size, a, b, x);
     }
     else {
        baseline (size, a, b, x);
     }
```

Le warmup permet de placer la machine en **régime permanent**, et donc par conséquent d'avoir des **mesures plus fiables et régulières**.

⇒ **Phase de mesure**

```
     /* measure repm repetitions */
     uint64_t t1 = rdtsc();
     for (i = 0; i < repm; i++)
        baseline (size, a, b, x);
     uint64_t t2 = rdtsc();
```

Suite à cela, une fois la machine en régime permanent, on effectue mesure le nombre de cycles, grâce à la fonction rdtsc, sur le nombre de répétitions défini.

⇒ **Calcul du nombre de cycle par itération**

```
     /* print performance */
     printf ("%.2f cycles/iterations\n",
             (t2 - t1) / ((float) size * repm));
             // or 2 * size * repm (equivalent
```

Une fois le nombre de cycle total récupéré, on souhaite récupérer le **nombre moyen de cycle par itération** de boucle.
Notre sujet étant composé de **deux boucles for successives**, effectuant chacune d'elles size itérations.
On divise donc le nombre total de cycle par le produit du nombre de répétitions effectuées et de la taille des tableaux (size).

1.2 Détermination des répétitions du noyau

1.2.1 Répétitions warmup

Le **warmup** permet **d'exclure le régime transitoire**, qui est un **obstacle** non-négligeable à la **précision des mesures**.
Il consiste en un **certain nombre de répétitions** (qui ne seront pas mesurées), utiles pour **« réchauffer » la machine**, et arriver ainsi dans le **régime permanent**.

Cette valeur **dépend de la configuration de la machine** utilisée, chaque membre du groupe a donc appliqué cette procédure, afin de déterminer son propre nombre de warmups nécessaires.

Afin de déterminer le nombre de warmups nécessaire à une mesure de performance cohérente et fiable, nous avons mis en place plusieurs scripts shell.

Pour cela, on va mesurer trois indicateurs de dispersions :

- l'écart-type
- (médiane – min) / min
- (max – médiane) / max

On estime que le nombre de warmups est suffisant dès lors que, si l'on considère les mesures obtenues, ces trois valeurs convergent, et soient relativement faibles (< à 5%).

Le script effectue donc un **balayage du nombre de warmups** (par exemple entre 0 et 1 000 000, avec un pas de 20 000), et pour chacune de ses valeurs, **exécute** le programme baseline de la même manière :

Pour chacun des appels, le **rapport de la médiane au minimum** (resp. médiane au maximum) est calculé (grâce à awk), et stocké dans un **fichier au format CSV** afin d'être exploité par la suite de manière graphique (courbe du rapport de la médiane au minimum en fonction du nombre de warmups).

Une fois la courbe tracée, il devient aisé de trouver le nombre de warmups nécessaire au bon fonctionnement de la simulation, en respectant la règle d'infériorité à 5 %.

Nous avons également trouvé judicieux d'**effectuer plusieurs balayages successifs**, en **réduisant** au fur et à mesure **les bornes** ainsi que **le pas**, en fonction des résultats précédents, afin d'être **le plus précis possible**.

Nous avons également trouvé judicieux de mesurer une **autre valeur statistique** : **l'écart-type.** De cette manière, nous obtiendrons deux courbes, que nous pourrons **croiser** afin de **déterminer le nombre optimal** de warmups.

En effet, l'écart-type représentant l'écart moyen entre les valeurs, il est important qu'il soit **le plus faible possible**.

1.2.2 Détermination de mesures

Les répétitions de mesures permettent **d'amortir l'erreur du timer**.

Toutefois, afin de **ne pas fausser les résultats** (qui sont en fait une moyenne de toutes les mesures), il est **important que cette valeur ne soit pas trop élevée**.

Aussi, sur conseil de M. OSERET, nous avons choisi d'effectuer **45 mesures** pour chaque méta-répétitions.

1.3 Protocole de mesure

Pour les mesures, nous avons mis en place deux scripts :

- measure.sh
- var_measure.sh

measure.sh permet de stocker les résultats d'une mesure dans un fichier passé en paramètre :

```
FILE="$1".csv
NB_WARMUPS=10000
NB_REPETS=45
VAL=3.14
TAILLE_TABS=25000

echo "Valeur" > $FILE
taskset -c 0 ./baseline $TAILLE_TABS $NB_WARMUPS $NB_REPETS $VAL | sed 's/[^0-9\.]*//g' >> $FILE
```

var_measure.sh permet lui d'effectuer des mesures en faisant varier la taille des tableaux.
En effet, il nous a été conseillé d'effectuer des mesures pour différents niveaux de remplissage de cache, afin de faire apparaître graphiquement le changement de cache effectué par la machine, selon la taille de données.

Les fichiers .csv générés par les deux scripts seront ensuite exploités par un tableur, en l'occurrence LibreOffice Calc, afin de tracer des courbes, de calculer divers indicateurs statistiques (médiane, moyenne, écart-type, etc.).

2 Travaux sur le cache L1 – Loïs JULLIEN

2.1 Configuration

2.1.1 Système

Les tests ont été effectués sous Ubuntu 18.04.1, installé en natif sur un MSI GP62 2QD Léopard . Pour plus de fiabilité, les tests ont été réalisés avec la machine branchée sur secteur, et le turbo-boost désactivé.

Processeur	Intel Core i5-4210H
Nombre de cœurs	2
Nombre de threads	4
Fréquence	2,90 Ghz
Taille du cache L1	32 kB (par cœur)
Taille du cache L2	256 kB (par cœur)
Taille du cache L3	3 MB
Taille de la RAM	4Go

Tableau 1: Informations du processeur

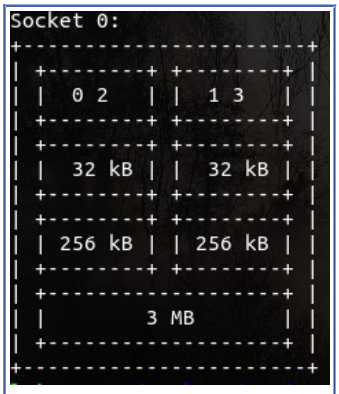


Illustration 2: Représentation du processeur (via likwid-topology)

2.1.2 Taille de données manipulées

On souhaite travailler sur un cache L1 de taille 32 kB.
Afin de s'assurer qu'on est bien dans ce cache, il est préférable de le remplir à la moitié de sa capacité, et de la même manière, afin de s'assurer qu'on ne sorte pas de ce cache, il convient de s'assurer que l'on ne dépasse pas 90 % de ce cache.
On a donc :

$$16\,000 < T_{L1} < 30\,400$$

Or, dans notre sujet, nous manipulons deux tableaux (de même taille, notée n) à une dimension de flottants.

Un flottant étant codé sur 4 octets, il vient :

$$(16\,000 < 2 * n * 4 < 30\,400) \Leftrightarrow 2000 < n < 3800$$

Pour la suite du sujet, on prendra donc **n = 2500**.

2.1.3 Calcul du nombre de warmups

Pour déterminer le nombre de warmups nécessaire pour L1, on va calculer l'indicateur de dispersion sur les nombre de cycles des méta-répétitions en fonction de l'écart-type.
Une première courbe représentant cet indicateur de dispersion en fonction du nombre de warmups a été tracé afin de fixer le nombre de warmups que l'on va utiliser pour la suite du projet.

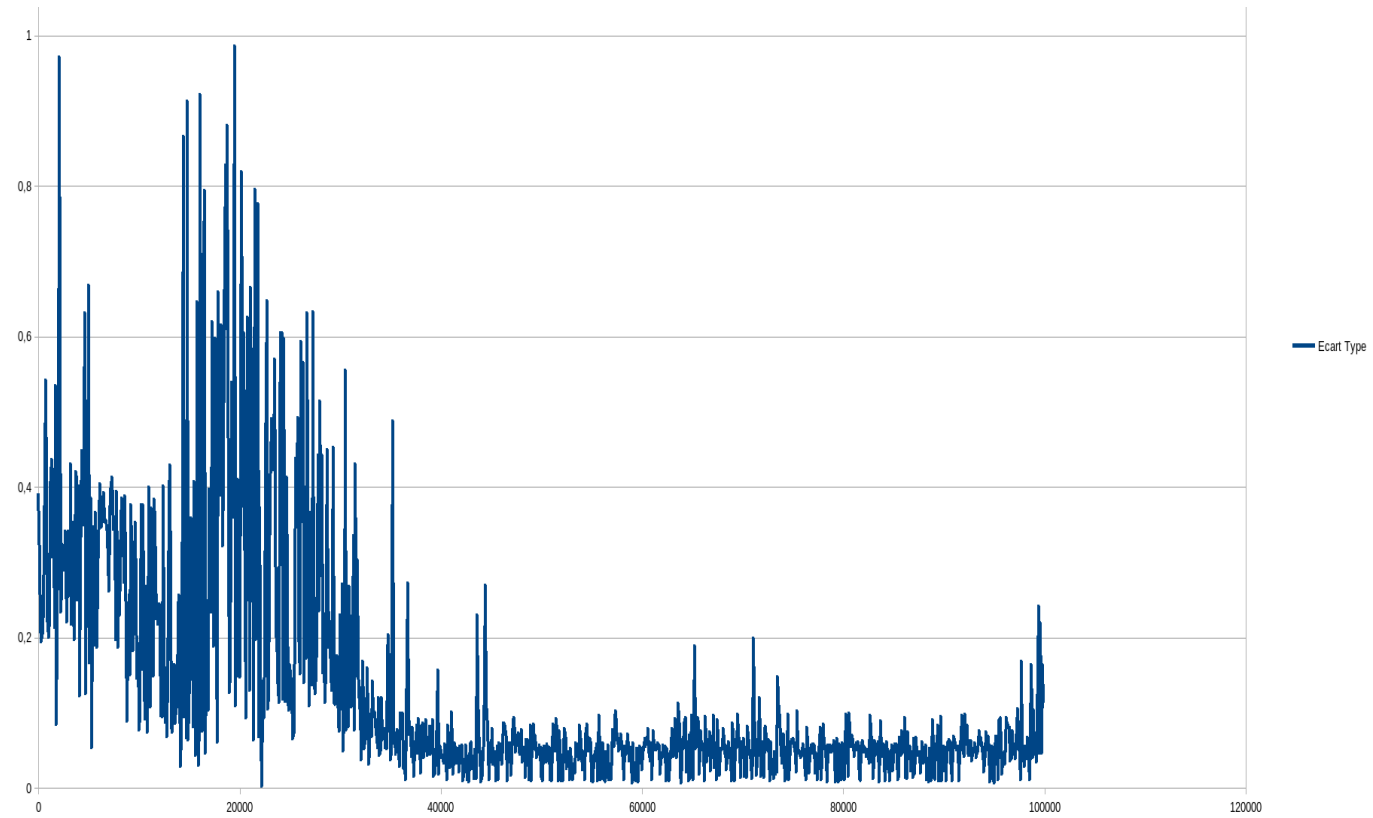


Illustration 3: Courbe de l'écart-type pour des warmups allant de 0 à 100 000

Cette courbe représentante la valeur de l'écart-type du cache L1 pour des warmups allant de 0 à 100 000. Un point est ajouté à la courbe tout les 50 warmups afin d'être le plus précis possible. Nous pouvons dans un premier temps remarquer que la valeur de l'écart-type à tendance à diminuer et à être relativement stable à partir de 50 000 warmups. Afin d'en être sur, nous allons diminuer le pas et voir plus large en étendant notre courbe à 4 millions de warmups.

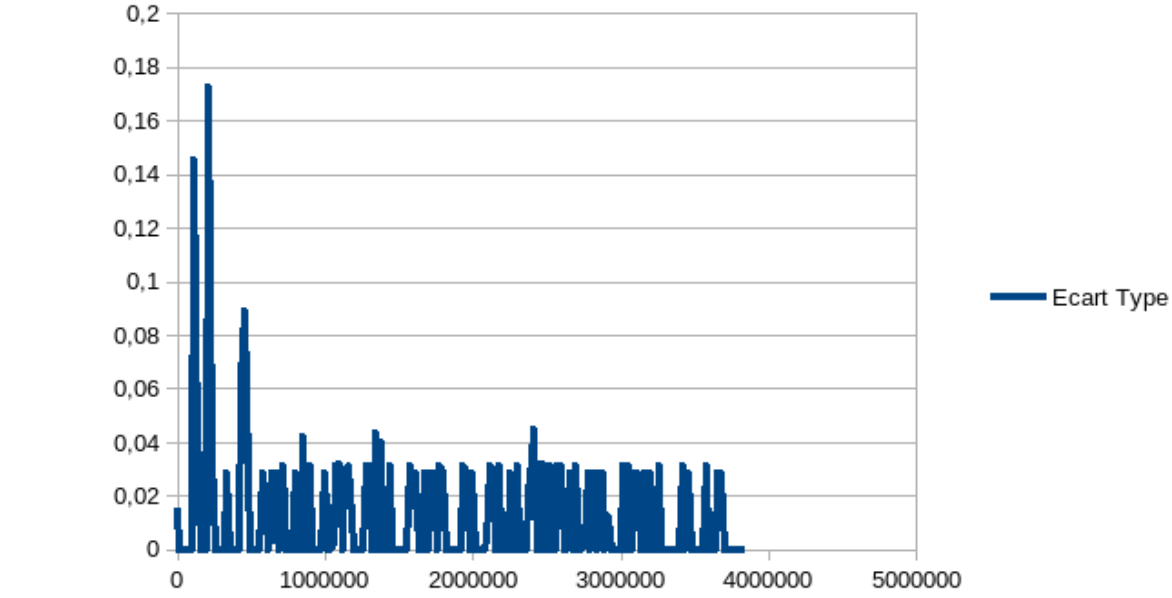


Illustration 4: Courbes de l'écart-type en fonction du nombre de warmups

Nous remarquons qu'il y a encore de grandes fluctuations jusqu'à 500 000 warmups, nous préférons donc prendre large et ainsi assurer un bon fonctionnement du programme. On peut voir sur la courbes qu'en choisissant un nombre de warmups supérieur à 1 000 000, l'écart-type est toujours inférieur à 5%. Nous allons donc fixer le nombre de warmups pour l'étude du cache L1 à 1 000 000.

2.2 Mesures

2.2.1 Options de compilation

Nous allons comparer les moyennes du nombre de cycles par itération avec différentes options de compilation. Nous allons utiliser le compilateur gcc avec les options -o2, -o3, -o3 -march=native, -ofast, -ofast -march=native, funrol-loops -march=native, floop-parallelize-all ainsi que le compilateur icc avec les options -o2, -o3 et -o3 -xHost.

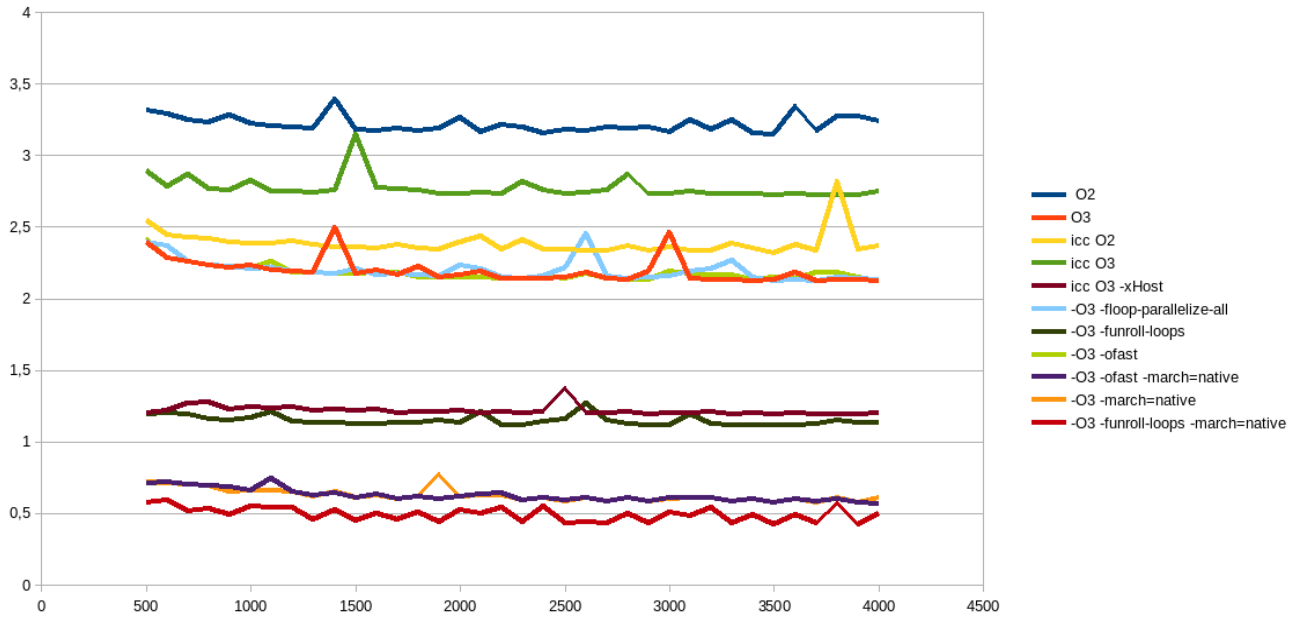


Illustration 3: courbe représentative des cycles par ittérations en fonction de la taille du tableau pour différents types de compilations

Le cache L1 est compris pour des tailles de tableaux entre 2000 et 3800, il faut donc se concentrer sur cette partie. Nous pouvons tout d'abord observer que l'option 02 est la courbe la plus élevée et que toutes les autres options diminuent les cycles/itération.
La compilation avec icc (en vert jaune et violet sur la courbe) permet un gain de rapidité mais est largement dépassé par -O3 funrolloops -march=native ici en rouge ne dépassant pas les 0,6.

2.2.2 Étude des exécutables avec Maqao

2.2.2.1 Compilateur GCC

Nous avons débuté l'analyse avec Maqao en utilisant un exécutable produit avec une compilation gcc -O2. La complexité du flux est mauvaise, et Maqao recommande l'utilisation de l'option de compilation -funroll-loops. Le programme passe 99,71 % du temps dans des boucles, qui pour la partie kernel ne sont pas du tout vectorisées dans le fichier kernel.c. Maqao annonce un speed-up de 7,84 pour une vectorisation totale. Le programme prend 27,88 secondes à s'exécuter. Tous les speed-ups seront calculés basé sur ce temps d'exécution.

Global Metrics

Total Time (s)		27.88
Time in loops (%)		99.71
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.06
Clean	Potential Speedup	1.18
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.84
	Nb Loops to get 80%	2

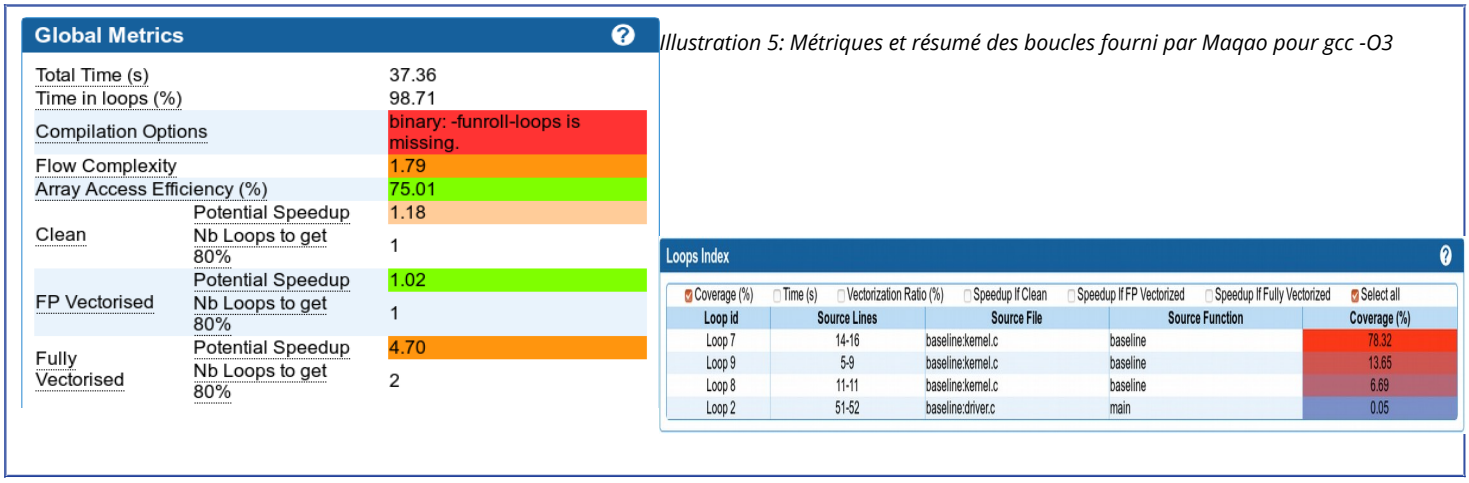
Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 5	14-16	baseline:kernel.c	baseline	50.14
Loop 6	5-11	baseline:kernel.c	baseline	49.35
Loop 2	51-52	baseline:driver.c	main	0.22

Illustration 4: Métriques et résumé des boucles fourni par Maqao pour gcc -O2

Nous avons ensuite étudié le résultat de la compilation avec gcc -O3. Là encore, Maqao recommande l'option -funroll-loops et la complexité du flux est de 1,79. La vectorisation des boucles est encore de 0 % pour les boucles de kernel.c, et le programme aura les mêmes métriques qu'avec -O2. Il prends plus de temps car nous avons augmenté le nombre de répétitions car sinon Maqao affichait un warning pour annoncé que le temps et trop court et non représentantatif.



Global Metrics

Total Time (s)		37.36
Time in loops (%)		98.71
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.79
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.18
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.70
	Nb Loops to get 80%	2

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 7	14-16	baseline:kernel.c	baseline	78.32
Loop 9	5-9	baseline:kernel.c	baseline	13.65
Loop 8	11-11	baseline:kernel.c	baseline	6.69
Loop 2	51-52	baseline:driver.c	main	0.05

Illustration 5: Métriques et résumé des boucles fourni par Maqao pour gcc -O3

Executons maintenant -O3 -floop-parallelize-all et -O3 -xHost :

Global Metrics

Total Time (s)		37.08
Time in loops (%)		98.7
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.80
Array Access Efficiency (%)		75.04
Clean	Potential Speedup	1.18
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.74
	Nb Loops to get 80%	2

Global Metrics

Total Time (s)		20.66
Time in loops (%)		99.13
Compilation Options		OK
Flow Complexity		1.01
Array Access Efficiency (%)		74.56
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	5

(%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	78.64
baseline:kernel.c	baseline	13.48
baseline:kernel.c	baseline	6.42
baseline:driver.c	main	0.16

(%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	88.77
baseline:kernel.c	baseline	9.49
baseline:kernel.c	baseline	0.58
baseline:kernel.c	baseline	0.29

Il manque encore des options pour floop-parallelize-all, le speed-up possible est encore à 1,18 et 4,74 pour la vectorisation, ce n'est don toujours pas la bonne solution.

-03 -xHost est en grande amélioration par rapport aux précédents. Plus aucune otion n'est demandé en plus, la complexité est de 1,01

2.2.2.2 Compilateur ICC

Exécutons avec icc -O2 et icc -O3

Global Metrics

Total Time (s)		41.04
Time in loops (%)		99.32
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		2.79
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.15
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.62
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	5.44
	Nb Loops to get 80%	2

Global Metrics

Total Time (s)		47.42
Time in loops (%)		99.53
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.03
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.26
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.99
	Nb Loops to get 80%	1

o (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	88.94
baseline:kernel.c	baseline	10.28
baseline:kernel.c	baseline	0.1

o (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	90.55
baseline:kernel.c	baseline	8.77
baseline:driver.c	main	0.13
baseline:kernel.c	baseline	0.08

Nous remarquons que le flow complexity est plus bas dans O3 que O2 et que le potentiel de speed-up est plus élevé pour 02. Il manque encore des options de compilation afin d'optimiser. Il est quand même possible d'augmenter le speed-up de 1,26 et 1,99. Ce n'est donc toujours pas la bonne option.

Passons maintenant à -O3 -march=native et -O3 -ofast -march=native

Global Metrics

Total Time (s)		25.3
Time in loops (%)		93.36
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.58
Array Access Efficiency (%)		75.04
Clean	Potential Speedup	1.13
	Nb Loops to get 80%	5
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1

Global Metrics

Total Time (s)		25.44
Time in loops (%)		93
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.57
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.12
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1

o (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	54.47
baseline:kernel.c	baseline	22.37
baseline:kernel.c	baseline	16.36
baseline:driver.c	main	0.16

o (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Source File	Source Function	Coverage (%)
baseline:kernel.c	baseline	52.67
baseline:kernel.c	baseline	23.9
baseline:kernel.c	baseline	16.43

Nous pouvons constater un net progrès, la partie vectorisation ne peut pas être mieux, et il est impossible d'avoir un speed-up mieux que ça pour ces partie. Il ne reste donc plus que la partie « clean » qui est relativement les mêmes pour les 2. essayons maintenant la dernière option de compilation : -O3 -funroll-loops -march=native

Global Metrics

Total Time (s)		26.4
Time in loops (%)		91.07
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.11
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1

Maqao nous indique que tout est au vert, le potentiel de speed-up est à 1 pour toutes les parties, c'est donc la confirmation de la courbe, -O3 -funroll-loops -march=native est bien la meilleur option.

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Vectorization Ratio (%)
Loop 5	16-16	baseline:kernel.c	baseline	59.85	0
Loop 9	5-9	baseline:kernel.c	baseline	18.11	100
Loop 7	11-11	baseline:kernel.c	baseline	12.95	100
Loop 4	13-14	baseline:driver.c	main	0.08	25
Loop 2	51-52	baseline:driver.c	main	0.08	0

3 Travaux sur le cache L2 – Clément LEFEVRE

3.1 Configuration

3.1.1 Système

Les tests ont été effectués sous **Ubuntu** 18.14.1 LTS, installé en **natif** (via un dual-boot) sur un Macbook Pro mi-2012.
Pour plus de fiabilité, les tests ont été réalisés avec la **machine branchée** sur secteur, le **turbo-boost désactivé**, sans **aucune application en exécution** excepté le terminal.

Processeur	Intel Core i5 3210M
Nombre de cœurs	2
Nombre de threads	4
Fréquence	2.5 Ghz
Taille du cache L1	32 Ko (par cœur)
Taille du cache L2	256 Ko (par cœur)
Taille du cache L3	3 Mo
Taille de la RAM	16 Go

Tableau 2: Informations du processeur

Illustration 6: Représentation du processeur (via likwid-topology)

3.1.2 Taille de données manipulées

On souhaite travailler sur un cache L2 de taille 256 Ko.

Afin de s'assurer qu'on est bien dans ce cache, il est préférable de le **remplir à la moitié** de sa capacité, et de la même manière, afin de s'assurer qu'on ne sorte pas de ce cache, il convient de s'assurer que l'on **ne dépasse pas 90 %** de ce cache.
On a donc :

$$128\,000 < T_{L2} < 230\,400$$

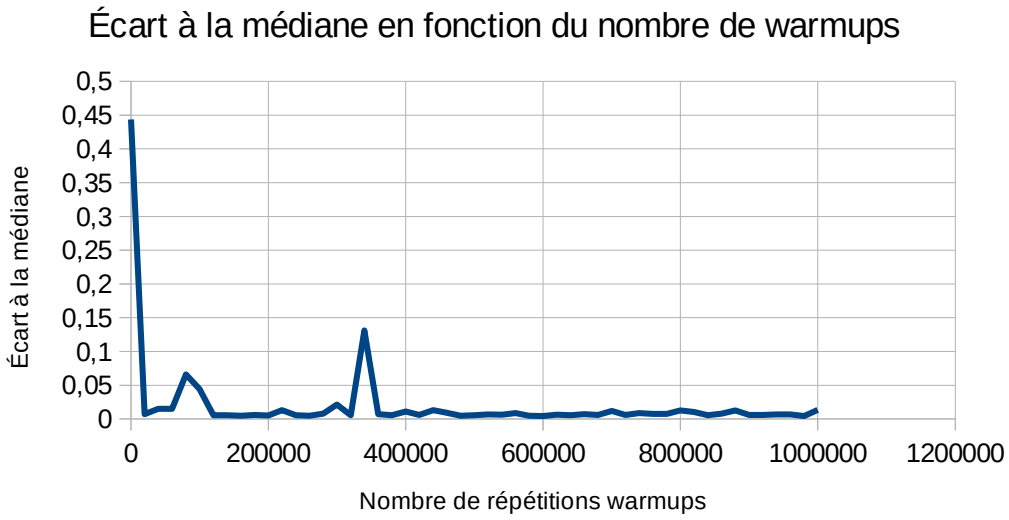
Or, dans notre sujet, nous manipulons deux tableaux (de même taille, notée n) à une dimension de flottants.
Un flottant étant codé sur 4 octets, il vient :

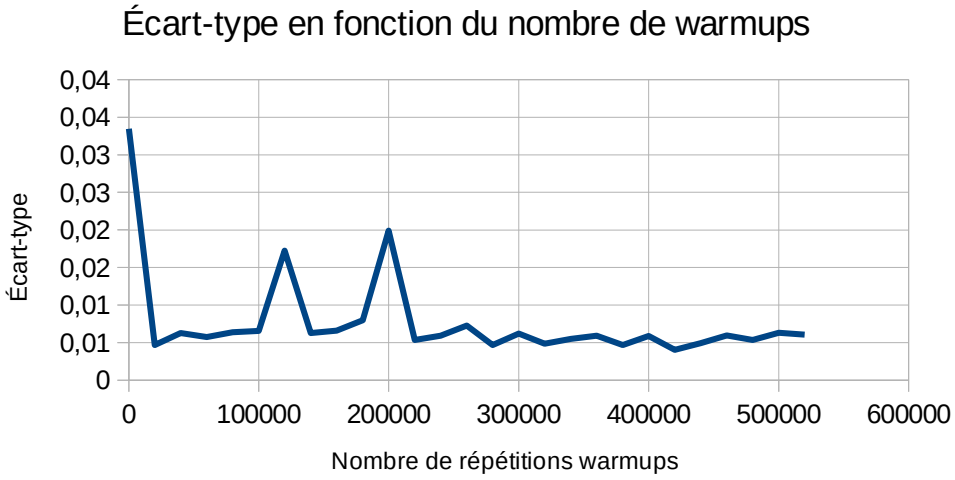
$$(128\,000 < 2 * n * 4 < 230400) \Leftrightarrow 16000 < n < 28800$$

Pour la suite du sujet, on prendra donc **n = 25000**.

3.1.3 Calcul du nombre de warmups

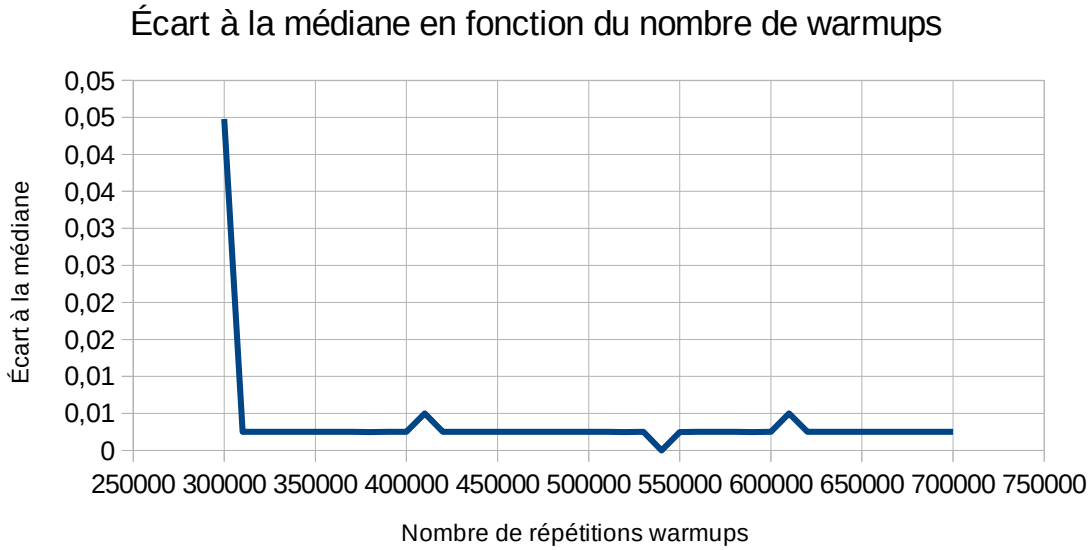
En utilisant les scripts mentionnés précédemment, j'ai obtenu les courbes suivantes :

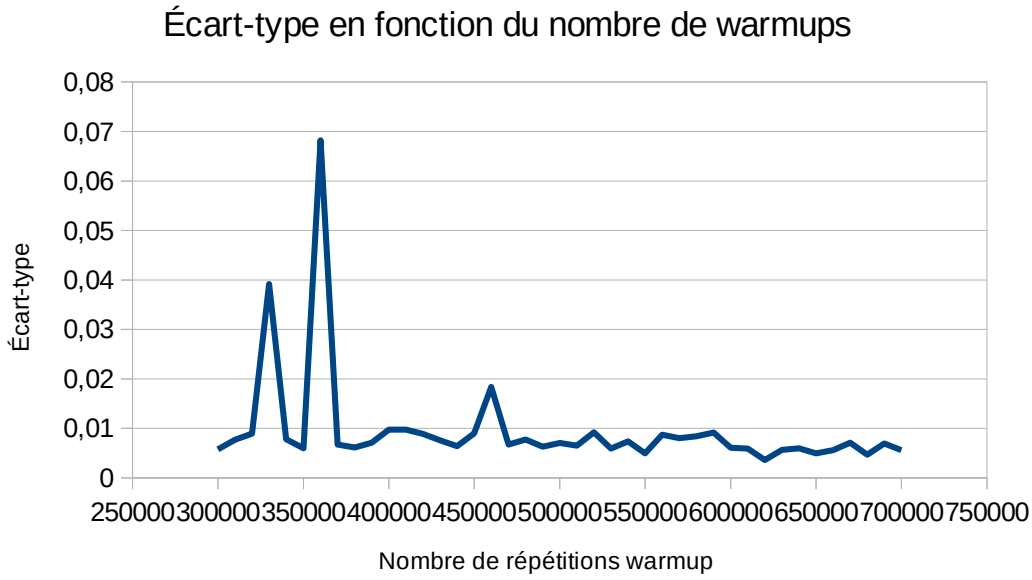




Si l'on considère uniquement la courbe de l'écart-type, on pourrait penser que le nombre optimal de warmups se situe aux alentours de 300 000, néanmoins, en croisant les résultats avec la courbe de l'écart à la médiane, on remarque qu'il demeure quelques instabilités.

Pour plus de précision, j'ai choisi d'effectuer un second balayage, entre 300 000 et 700 000 répétitions. On obtient les courbes suivantes :





Cette fois ci, en croisant les résultats, on peut être plus précis.
C'est pourquoi, pour la suite des mesures, on prendra effectuera **500 000 répétitions warmups**.

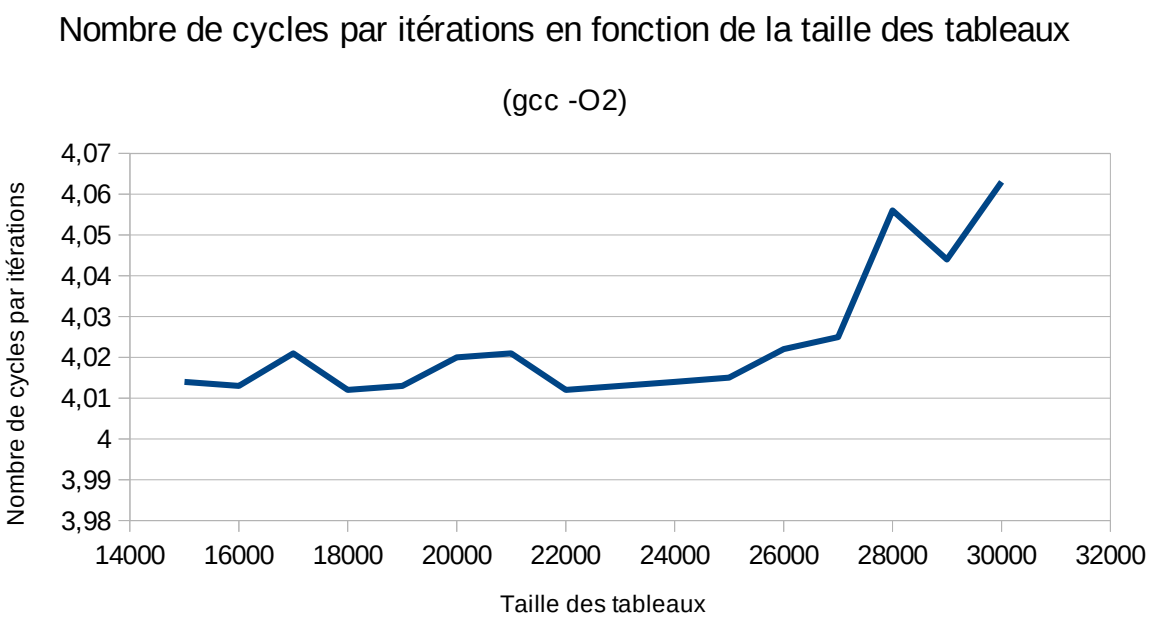
3.2 Mesures

3.2.1 GCC -O2

Avec ce degré d'optimisation, gcc exécute presque la totalité des optimisations prises en charges, sans compromis sur la vitesse et l'espace utilisé.
Pour n = 25 000, valeur de référence pour le cache de niveau L2, on obtient :

Moyenne (cycles/itérations)	4,02258064516129
Minimum (cycles/itérations)	4,01
Maximum (cycles/itérations)	4,04
Médiane (cycles/itérations)	4,02
Écart-type (cycles/itérations)	0,00773207257057

Il peut également être judicieux de mesurer le nombre de cycle par itérations en faisant varier la taille des tableaux (n), afin de visualiser ce qui se produit lorsque le cache L2 est rempli.



Comme attendu, dès lors que l'on approche de la borne supérieure calculée (28800), le cache est rempli, l'accès aux données est donc plus long.
Il se peut également que des données soient déplacées dans le cache L3, ce qui expliquerait cette augmentation de nombre de cycles par itérations.

Lorsque l'on analyse le programme, à l'aide de MAQAO, on obtient le résultat suivant :

Global Metrics

Total Time (s)		20.3
Time in loops (%)		99.71
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	6.86
	Nb Loops to get 80%	2

Comme on pouvait s'y attendre, la majorité du temps d'exécution est passé dans les boucles.

MAQAO nous indique également que la complexité des flux est trop élevée.

De plus, en étant totalement vectorisé, on pourrait avoir un speed-up de 6.86.

Si l'on s'intéresse de plus prêt aux boucles, on remarque que les deux boucles à optimiser sont celles du noyau. En effet, à elles-deux, elles couvrent 99.71 % du temps d'exécution du programme.
On remarque qu'aucune de ces boucles n'est vectorisée (0%), et que le speed-up de chacune d'elles serait important (entre 6.5 et 7.5) si c'était le cas.

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 5	13-15	baseline:kernel.c	baseline	50.74	10.3	0	1	1	7.43
Loop 6	4-10	baseline:kernel.c	baseline	48.97	9.94	0	1.25	1.6	6.46

MAQAO nous conseille donc d'autoriser cette vectorisation de boucle (via -O3 ou -Ofast) par exemple.

Workaround

- Try another compiler or update/tune your current one:
  - recompile with ftree-vectorize (included in O3) to enable loop vectorization and with fassociative-math (included in Ofast or ffast-math) to extend vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

3.2.2 GCC -O3

Ce degré d'optimisations ajoute encore quelques flags à ceux déjà présents dans -O2.
Cependant, la taille du code généré est assez fortement impactée.
Pour n = 25 000, on a :

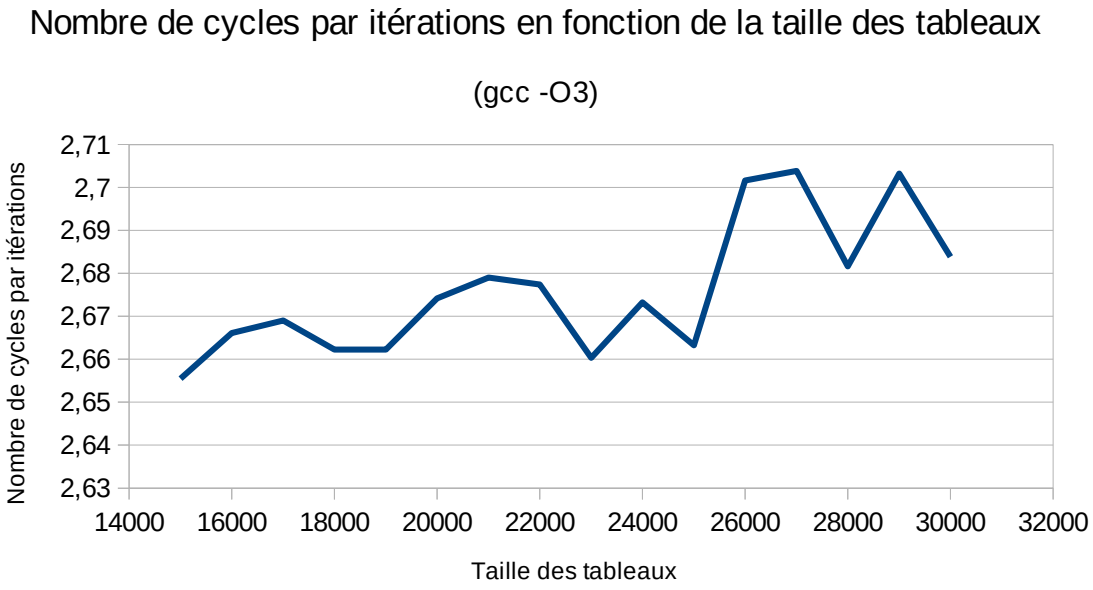
Moyenne (cycles/itérations)	2,68290322580645
Minimum (cycles/itérations)	2,68
Maximum (cycles/itérations)	2,69
Médiane (cycles/itérations)	2,68
Écart-type (cycles/itérations)	0,004614143753791

On observe un gain de performance non-négligeable, en effet, on a :

$$G_{moyen}(gcc-O3/gcc-O2)\approx\frac{4.023}{2.683}\approx 1.5$$

$$G_{median}(gcc-O3/gccO2)=\frac{4.02}{2.68}=1.5$$

Si l'on mesure le nombre de cycle par itérations en faisant varier la taille des tableaux, on obtient la courbe suivante :



Comme pour -O2, on remarque que dès lors que le cache L2 est presque rempli, le nombre de cycles par itérations augmente.

Lorsque l'on analyse l'exécution du programme avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		13.48
Time in loops (%)		99.7
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.74
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.39
	Nb Loops to get 80%	2

On remarque que le temps total d'exécution est bien plus court qu'avec -O2 (13.48s contre 20.3).

De la même manière, la complexité des flux a légèrement baissé, de même que le speed-up potentiel si les boucles étaient totalement vectorisées.

Si l'on s'intéresse aux boucles, on remarque que le compilateur a divisé la première boucle en deux, et a vectorisé les deux boucles obtenues.

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 7	13-15	baseline:kernel.c	baseline	73.29	9.88	0	1	1	7.43
Loop 9	4-8	baseline:kernel.c	baseline	15.13	2.04	100	1	1	1.18
Loop 8	10-10	baseline:kernel.c	baseline	11.28	1.52	100	1	1.56	2

En effet, en étudiant de plus près la transformation, MAQAO nous indique le code source

```
10:                     b[i] = a[i] + x;
```

a été remplacé par le code assembleur suivant :

```
0xc90 MOVAPS (%R10,%RAX,1),%XMM1
0xc95 ADD $0x1,%ECX
0xc98 ADDPS %XMM2,%XMM1
0xc9b MOVUPS %XMM1,(%R8,%RAX,1)
0xca0 ADD $0x10,%RAX
0xca4 CMP %ECX,%R12D
0xca7 JA c90 <baseline+0x110>
```

La seconde boucle n'a elle pas été optimisée (0 % de vectorisation).

3.2.3 FCC -O3 -march=native

Le flag -march permet d'indiquer au compilateur quel code il doit générer selon l'architecture du processeur. Assigner ce flag au mot clé native permet de lui indiquer qu'il doit déterminer par lui-même cette architecture.

Pour n = 25 000 on a :

Moyenne (cycles/itérations)	1,17129032258065
Minimum (cycles/itérations)	1,17
Maximum (cycles/itérations)	1,18
Médiane (cycles/itérations)	1,17
Écart-type (cycles/itérations)	0,003407771005482

Si l'on compare ces résultats à ceux obtenus via -O3, on a :

$$G_{moyen}(gcc-O3/gcc-O3-march=native)\approx\frac{2.683}{1.171}\approx 2.29$$

$$G_{median}(gcc-O3/gcc-O3-march=native)=\frac{2.68}{1.17}\approx 2.29$$

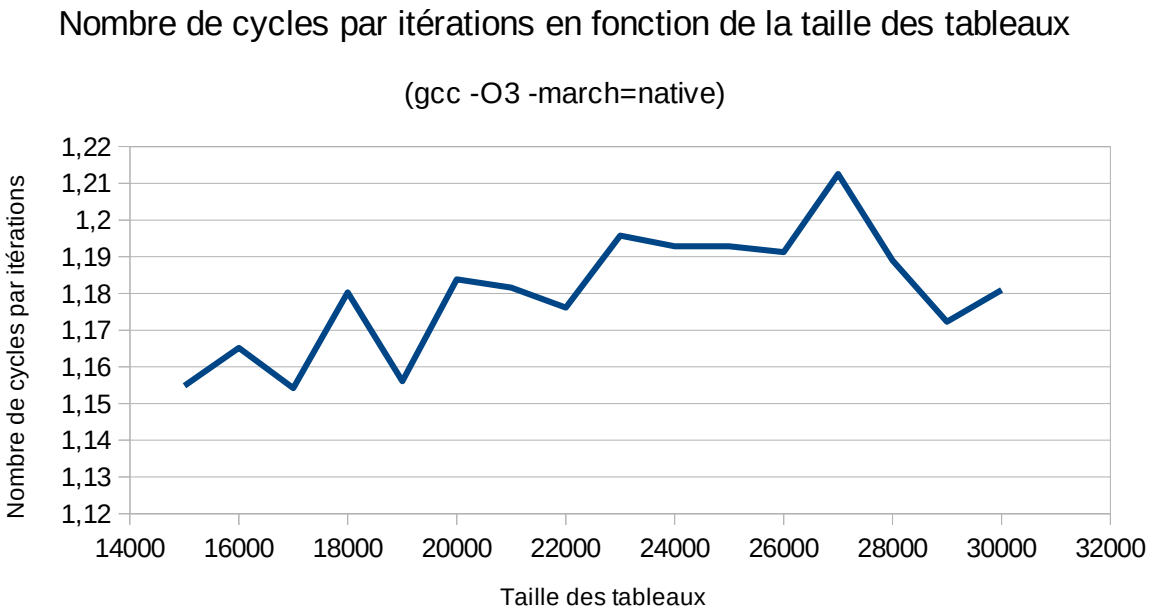
Et par rapport à O2 :

$$G_{moyen}(gcc-O3-march=native/gcc-O2)\approx\frac{4.023}{1.171}\approx 3.44$$

$$G_{median}(gcc-O3-march=native/gcc-O2)=\frac{4.02}{1.17}\approx 3.44$$

Le gain de performances apporté par cette optimisation est donc réellement important.

En faisant varier la taille des tableaux, on peut effectuer le même constat que précédemment, à savoir que le nombre de cycle par itérations augmente plus le cache est proche d'être totalement rempli.



Lorsque l'on analyse le résultat de l'exécution avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		59.9
Time in loops (%)		99.47
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.41
Array Access Efficiency (%)		75.02
Clean	Potential Speedup	1.42
	Nb Loops to get 80%	5
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.10
	Nb Loops to get 80%	2

La complexité des flux a encore diminué, de même que les potentiels speed-up obtenus après vectorisation.

On remarque également que le flag -funroll-loops est manquant, et qu'il pourrait apporter un gain de performances important au programme.

Concernant les boucles, de la même manière que pour -O3, la première boucle a été scindée en deux.
(Une quatrième boucle issue du driver apparaît dans la synthèse, mais elle n'est pas importante).

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 7	13-15	baseline:kernel.c	baseline	40.93	24.52	87.5	1.42	NA	NA
Loop 9	4-8	baseline:kernel.c	baseline	32.59	19.52	100	1.5	1	1.2
Loop 8	10-10	baseline:kernel.c	baseline	25.88	15.5	100	1.5	1	1.17
Loop 2	51-52	baseline:driver.c	main	0.07	0.04	0	3	1	8

Grâce au flag -march=native, la boucle n°7, a elle aussi été vectorisée (à 87.5%).
Le speed-up potentiel diminue, ce qui montre que nous atteignons quasiment les limites d'optimisations pour le programme (concernant la partie compilation).

On remarque également que le code assembleur généré diffère. En effet, comme mentionné précédemment, le flag -march=native permet de mettre en place des optimisations propres au processeur, ce qui explique cela.
Par exemple, pour la boucle 8, on a :

GCC -O3	GCC -O3 -march=native
0xc90 MOVAPS (%R10,%RAX,1),%XMM1 0xc95 ADD $0x1,%ECX 0xc98 ADDPS %XMM2,%XMM1 0xc9b MOVUPS %XMM1,(%R8,%RAX,1) 0xca0 ADD $0x10,%RAX 0xca4 CMP %ECX,%R12D 0xca7 JA c90 <baseline+0x110>	0xd10 VADDPS (%R9,%RAX,1),%YMM2,%YMM1 0xd16 ADD $0x1,%ECX 0xd19 VMOVUPS %XMM1,(%R8,%RAX,1) 0xd1f VEXTRACTF128 $0x1,%YMM1,0x10(%R8,%RAX,1 0xd27 ADD $0x20,%RAX 0xd2b CMP %ECX,%R12D 0xd2e JA d10 <baseline+0x190>

3.2.4 ICC -O2

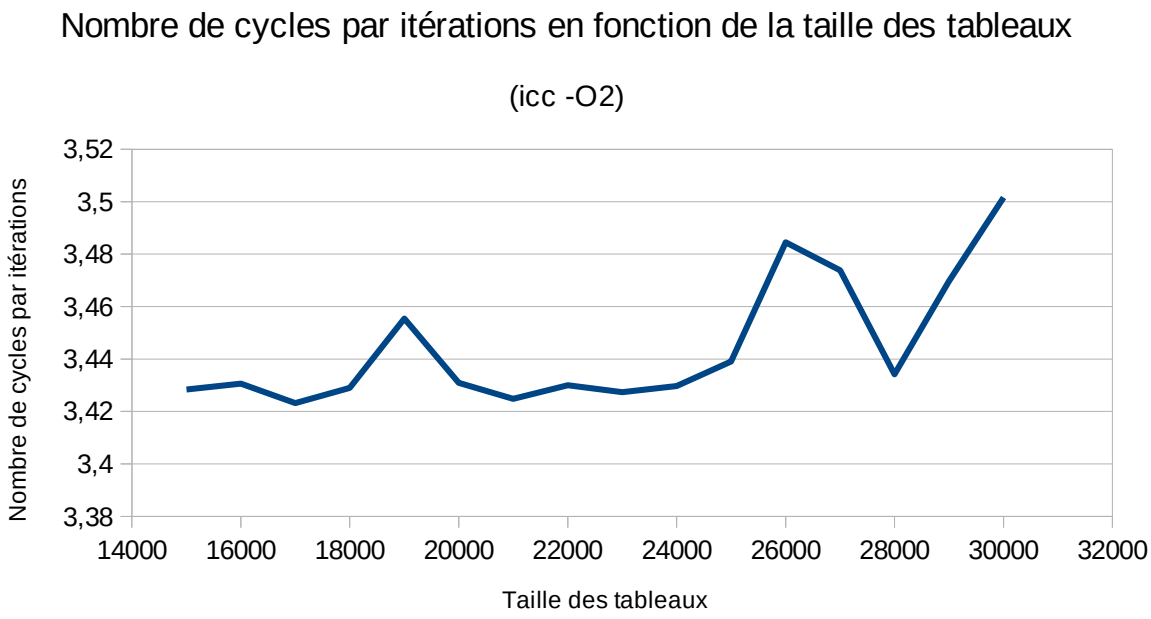
Ce niveau d'optimisation est celui recommandé par Intel au sein de sa documentation. Contrairement à celui de GCC, ce niveau d'optimisation autorise la vectorisation, ainsi que d'autres optimisations communes.
Pour n = 25000, on a :

Moyenne (cycles/itérations)	3,45870967741935
Minimum (cycles/itérations)	3,45
Maximum (cycles/itérations)	3,47
Médiane (cycles/itérations)	3,46
Écart-type (cycles/itérations)	0,004275461366037

On constate que le niveau 2 d'optimisations de icc est légèrement plus efficace que celui de gcc, puisqu'on a :

$$G_{moyen}(icc-O2/gcc-O2)\approx\frac{4.023}{3.459}\approx1.16$$

$$G_{median}(icc-O2/gcc-O2)=\frac{4.02}{3.46}\approx1.16$$



Si l'on analyse l'exécution avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		172.14
Time in loops (%)		99.87
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		2.75
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.77
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.24
	Nb Loops to get 80%	1

A la manière de GCC -O2, la complexité des flux est mauvaise.

MAQAO préconise également l'utilisation du flag -xHost, qui permet de mettre en place des optimisations propres au CPU.

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 5	4-10	baseline:kernel.c	baseline	87.17	150.06	0	1.22	1.33	7.27
Loop 7	13-15	baseline:kernel.c	baseline	12.69	21.84	100	1	1	1.5
Loop 2	51-52	baseline:driver.c	main	0.01	0.02	0	3	1	8

Contrairement à GCC -O2, où aucune des boucles n'était vectorisé, on remarque ici que la boucle n°7 l'est (avec un ration de 100%).
La faible performance vient donc de la boucle n°5, qui couvre à elle seule 87.17 % du temps d'exécution. Il s'agît de cette boucle :

```
4:      for (i = 0; i < n; i++) {
5:              if ((i < n/2) && (a[i] > x))
6:                      b[i] = a[i];
7:              else if (i < n/2)
8:                      b[i] = x;
9:              else
10:                     b[i] = a[i] + x;
11:     }
```

Dans les axes d'améliorations, MAQAO nous indique que l'on pourrait déplacer les expression conditionnelles (If hoisting).
Il nous indique également qu'en vectorisant cette boucle, on pourrait obtenir un speed-up potentiel de 5.82.

3.2.5 ICC -O3

L'option -O3 d'icc met en place les mêmes optimisations que -O2, et ajoute des optimisations plus agressives au niveau des boucles : déroulage, fusion, etc.

Pour n = 25 000, on a :

Moyenne (cycles/itérations)	3,67225806451613
Minimum (cycles/itérations)	3,67
Maximum (cycles/itérations)	3,68
Médiane (cycles/itérations)	3,67
Écart-type (cycles/itérations)	0,004250237185033

Contrairement à ce à quoi on pouvait s'attendre, l'option -O3 ralentit le programme par rapport à -O2. Ainsi :

$$G_{moyen}(icc-O3/icc-O2)\approx\frac{3.459}{3.677}\approx 0.94$$

$$G_{median}(icc-O3/icc-O2)=\frac{3.46}{3.67}\approx 0.94$$

En analysant l'exécution avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		184.16
Time in loops (%)		99.89
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.28
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.35
	Nb Loops to get 80%	1

La complexité du flux est optimale (1.00).

Le speed-up peut encore être amélioré, grâce à la vectorisation (environ 30 % de speed-up potentiel).

Si l'on s'intéresse aux boucles, on observe que les deux boucles principales sont vectorisées (à 100%), ce qui n'était pas le cas avec GCC.

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 10	2-10	baseline:kernel.c	baseline	87.98	162.02	100	1	1.33	1.33
Loop 6	13-15	baseline:kernel.c	baseline	11.87	21.86	100	1	1	1.5
Loop 2	51-52	baseline:driver.c	main	0.03	0.06	0	3	1	8
Loop 3	13-14	baseline:driver.c	main	0.01	0.02	25	1.5	2.13	7.11

Néanmoins, on remarque que c'est toujours la même boucle qui couvre la majorité du temps d'exécution (87.98%).

En cherchant plus de précisions, MAQAO nous indique que la boucle est vectorisée, mais qu'elle n'occupe que 62% de la taille du registre, et qu'en la vectorisant totalement, on obtiendrait un speed-up de 1.33.

MAQAO réclame également le flag speed-up, qui permettrait à ICC d'effectuer de meilleurs optimisations, adaptées au CPU, c'est donc ce que nous allons effectuer au sein de la prochaine partie.

3.2.6 ICC -O3 -xHost

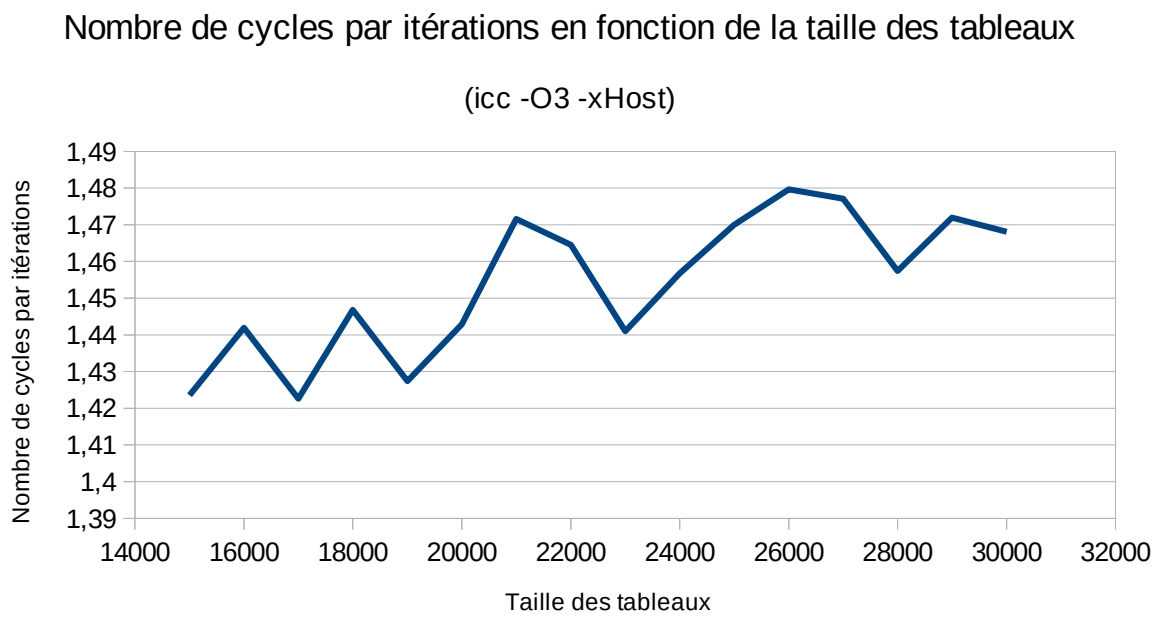
Le flag -xHost d'icc est semblable au -march de GCC, puisqu'il permet au compilateur de récupérer les informations concernant le processeur, et de mettre en place un certain nombre d'optimisations spécifiques à ce processeur.
Pour n = 25 000 on a :

Moyenne (cycles/itérations)	1,46064516129032
Minimum (cycles/itérations)	1,46
Maximum (cycles/itérations)	1,47
Médiane (cycles/itérations)	1,46
Écart-type (cycles/itérations)	0,002497310381147

On remarque que le flag -xHost améliore considérablement la performance.

$$G_{moyen}(icc-O3-xHost/icc-O3)\approx\frac{3.672}{1.460}\approx 2.51$$

$$G_{median}(icc-O3-xHost/icc-O3)=\frac{3.67}{1.46}\approx 2.51$$



En analysant le résultat avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		72.52
Time in loops (%)		99.76
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		80.83
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	1

Tous les indicateurs sont au vert, le speed-up obtenu ne peut être amélioré (i.e. légèrement si tout est vectorisé : 1.04).

Si l'on s'intéresse aux boucles, on observes le résultat suivant :

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 10	4-10	baseline:kernel.c	baseline	69.8
Loop 6	13-15	baseline:kernel.c	baseline	29.76
Loop 12	4-10	baseline:kernel.c	baseline	0.11
Loop 8	13-15	baseline:kernel.c	baseline	0.03
Loop 11	4-10	baseline:kernel.c	baseline	0.03
Loop 4	13-14	baseline:driver.c	main	0.03

On remarque qu'il y a de nombreuses lignes, ce qui peut paraître troublant à première vue.
Il semblerait que le compilateur ait généré de petites boucles. En s'y intéressant de plus près, MAQAO nous indique : « *It is peel/tail loop of related source loop which is unrolled by 16 (including vectorization).* ».
Les deux boucles principales ont donc été déroulées par le compilateur, et les autres boucles générées sont en fait des boucles résiduelles de ce déroulage.

On remarque que la boucle n°10 demeure celle couvrant le plus du temps d'exécution.
Néanmoins, en l'analysant, MAQAO nous indique qu'on ne pourrait obtenir qu'un speed-up de 1.06 en la vectorisant encore plus (91 % de la taille des registres est utilisée).

3.2.7 GCC -Ofast

-Ofast est le degré d'optimisation supérieur à -O3. Néanmoins, ce degré ignore les normes strictes de conformités.
Pour n = 25 000, on a :

Moyenne (cycles/itérations)	2,6758064516129
Minimum (cycles/itérations)	2,67
Maximum (cycles/itérations)	2,71
Médiane (cycles/itérations)	2,67
Écart-type (cycles/itérations)	0,00847513757937

On remarque qu'il n'y a pas de réel impact sur le temps d'exécution, comparé à -O3, les résultats sont équivalents.

En analysant avec MAQAO, on obtient :

Global Metrics

Total Time (s)		134.08
Time in loops (%)		99.75
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.75
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.49
	Nb Loops to get 80%	2

Les résultats sont semblables à ceux observés pour GCC -O3.
En effet, il y a peu de différences entre les optimisations de -O3 et celles de -Ofast. Cependant, -Ofast met en place des optimisations assez agressives, qui peuvent nuire au bon fonctionnement du programme.

Concernant les boucles, on a :

Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 7	13-15	baseline:kernel.c	baseline	74.63	100.06	0	1	1	7.43
Loop 9	4-8	baseline:kernel.c	baseline	14.16	18.98	100	1	1	1.18
Loop 8	10-10	baseline:kernel.c	baseline	10.95	14.68	100	1	1.56	2
Loop 4	13-14	baseline:driver.c	main	0.01	0.02	25	1.5	2.13	7.11

Ceci est sensiblement identique à ce que l'on avait pour GCC -O3, avec une des boucles occupant la grande partie du temps d'exécution.
MAQAO nous indique qu'il serait préférable d'ôter les expression conditionnelles de cette boucle (« If hoisting »).

Try to simplify control inside loop: ideally, try to remove all conditional expressions, for example by (if applicable):

- hoisting them (moving them outside the loop)
- turning them into conditional moves, MIN or MAX

3.2.8 GCC -Ofast -march=native

De la même manière qu'on l'a fait pour gcc -O3, on peut ajouter le flag -march=native à -Ofast.
On observe les résultats suivants :

Moyenne (cycles/itérations)	1,16612903225806
Minimum (cycles/itérations)	1,16
Maximum (cycles/itérations)	1,18
Médiane (cycles/itérations)	1,17
Écart-type (cycles/itérations)	0,006152191671721

Les résultats fournis par MAQAO sont semblables à ceux de GCC -O3 -march=native.

3.2.9 GCC -O3 -funroll-loops

-funroll loops permet d'indiquer au compilateur de dérouler les boucles dont il peut prévoir le nombre d'itérations à l'avance. Il n'est pas activé dans les différents degré -O.
Pour n = 25 000 :

Moyenne (cycles/itérations)	2,11870967741935
Minimum (cycles/itérations)	2,13
Maximum (cycles/itérations)	2,11
Médiane (cycles/itérations)	2,12
Écart-type (cycles/itérations)	0,07839182938273

En analysant l'exécution avec MAQAO, on obtient la synthèse suivante :

Global Metrics

Total Time (s)		105.78
Time in loops (%)		99.52
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.01
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.05
	Nb Loops to get 80%	2

On remarque que tous les voyants sont au vert, la complexité des flux est optimale, et d'après les indicateurs de speed-up potentiel, il est difficile de faire mieux.

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 5	13-15	baseline:kernel.c	baseline	73.71
Loop 9	4-8	baseline:kernel.c	baseline	13.09
Loop 7	10-10	baseline:kernel.c	baseline	12.68

Concernant les boucles, MAQAO nous indique qu'elles sont déroulées par 4 par le compilateur.

Je ne suis pas parvenu à accéder à l'analyse détaillée de la première boucle, MAQAO renvoyant l'erreur suivante : « *This loop cannot be analyzed: too many paths (256 paths > max_paths=4). Rerun with -max_paths=256.* »

Cependant, même en ajoutant cette option à la commande, l'erreur persistait.

3.2.10 GCC -O3 -funroll-loops -march=native

Au vu des expérimentations précédentes, j'ai pu remarquer que trois flags avaient une influence importante sur la performance du programme : -O3, -funroll-loops et -march=native.
Ainsi, en cumulant ces trois options, on obtient les résultats suivants :

Moyenne (cycles/itérations)	0,963870967741936
Minimum (cycles/itérations)	0,95
Maximum (cycles/itérations)	0,99
Médiane (cycles/itérations)	0,96
Écart-type (cycles/itérations)	0,008032193289025

Il s'agît de l'option la plus performance, puisque l'on passe sous la barre des 1 cycles par itérations.

Global Metrics

Total Time (s)		50.28
Time in loops (%)		99.29
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.02
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.08
	Nb Loops to get 80%	1

Lorsque l'on analyse l'exécution avec MAQAO, on obtient des résultats assez proches de ceux obtenus via GCC -O3 -march=native.

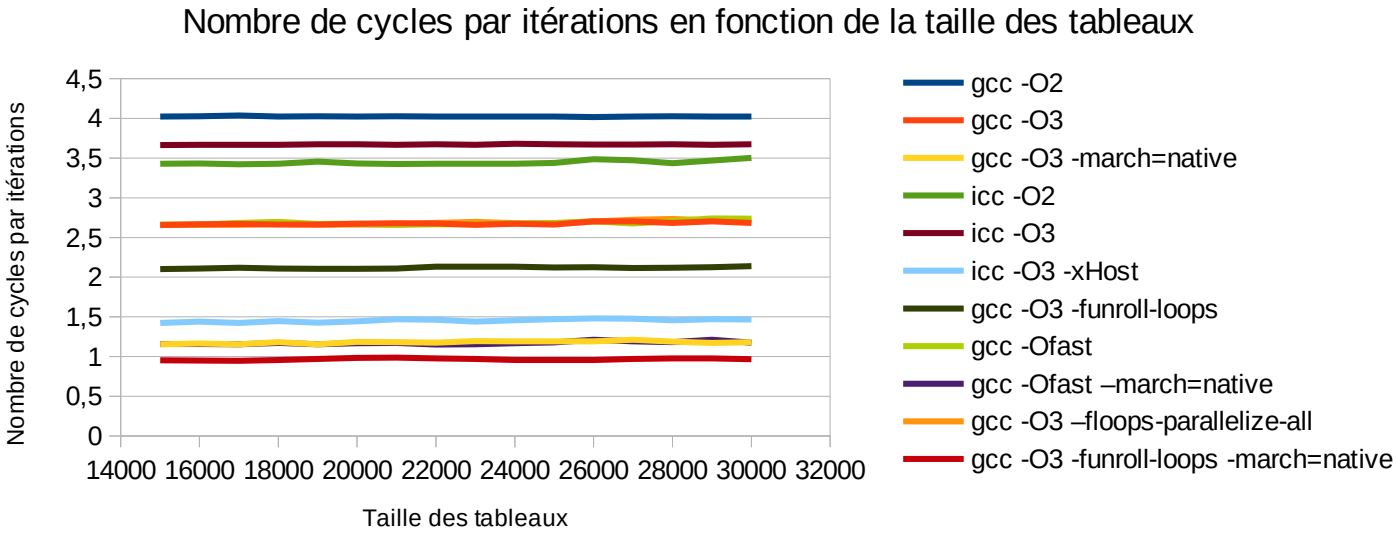
Loops Index

Coverage (%) Time (s) Vectorization Ratio (%) Speedup If Clean Speedup If FP Vectorized Speedup If Fully Vectorized Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 5	15-15	baseline:kernel.c	baseline	41.41	20.82	0	0	0	0
Loop 7	10-10	baseline:kernel.c	baseline	30.71	15.44	100	1	1	1
Loop 9	4-8	baseline:kernel.c	baseline	27.13	13.64	100	1	1	1.38
Loop 2	51-52	baseline:driver.c	main	0.04	0.02	0	3.22	1	8

3.3 Conclusion

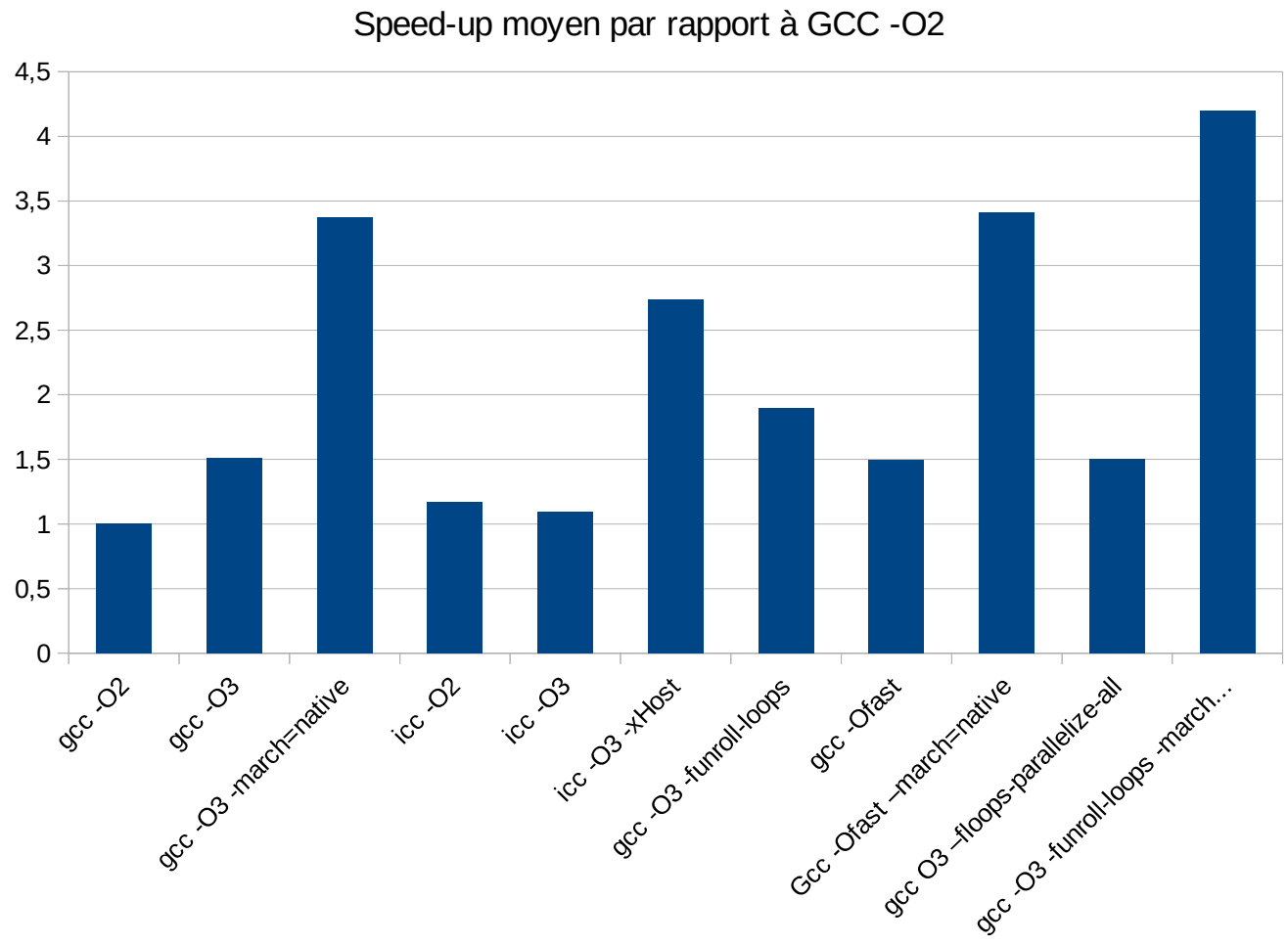
Pour conclure, on peut dans un premier temps regarder ce graphique, comparant la performance de toutes les optimisations testées au sein de ce rapport :



La plus efficace est donc la dernière, gcc -O3 -funroll-loops -march=native.
En effet, -O3 permet d'effectuer la quasi-totalité des optimisations, -funroll-loops permet d'indiquer au compilateur de mettre en place du déroulage de boucle, et -march=native lui indique de mettre en place des optimisations propre au processeur.

C'est d'ailleurs ce type d'optimisations qui joue un rôle majeur pour l'amélioration de la performance du programme, puisque les trois options les plus efficaces possèdent tous une optimisation de ce type (-march=native pour GCC et -xHost pour ICC).

Il peut également être judicieux de comparer ces valeurs en les rapportant à notre option de référence, gcc -O2.
De cette manière, on obtient le speed-up de chaque option par rapport à gcc -02 :


Speed-up moyen par rapport à GCC -O2
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
gcc -O2
gcc -O3
gcc -O3 -march=native
icc -O2
icc -O3
icc -O3 -xHost
gcc -O3 -funroll-loops
gcc -Ofast
Gcc -Ofast –march=native
gcc O3 –floops-parallelize-all
gcc -O3 -funroll-loops -march...

4 Travaux sur le cache L3 – Étienne SAUVÉE

4.1 Configuration

4.1.1 Système

Les tests ont été effectués sous Debian 9.8 Stretch, installé en dual boot sur un MSI GP62MVR 7RF Léopard pro.
Pour plus de fiabilité, les tests ont été réalisés avec la machine branchée sur secteur, et le turbo-boost désactivé.

Processeur	Intel Core i5-7300HQ
Nombre de cœurs	4
Nombre de threads	4
Fréquence	2,5 GHz
Taille du cache L1	32 ko
Taille du cache L2	256 ko
Taille du cache L3	6 Mo
Taille de la RAM	8 Go

Tableau 3: Informations du processeur

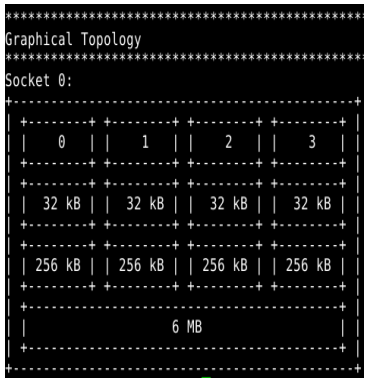


Illustration 7: Représentation du processeur (via likwid-topology)

4.1.2 Taille de données manipulées

On souhaite travailler sur un cache L3 de taille 6 Mo.

Afin de s'assurer qu'on est bien dans ce cache, il est préférable de dépasser 3 fois la capacité du cache L2, et de la même manière, afin de s'assurer qu'on ne sorte pas de ce cache, il convient de s'assurer que l'on ne dépasse pas 50 % de ce cache.
On a donc :

$$768\,000 < T_{L3} < 3\,000\,000$$

Or, dans notre sujet, nous manipulons deux tableaux (de même taille, notée n) à une dimension

de flottants.
Un flottant étant codé sur 4 octets, il vient :

$$(768\,000 < 2 * n * 4 < 3\,000\,000) \Leftrightarrow 96\,000 < n < 375\,000$$

Pour la suite du sujet, on prendra donc **n = 200 000**.

4.1.3 Calcul du nombre de warmups

Pour déterminer le nombre de warmups nécessaire, on va calculer différents indicateurs de dispersion sur les nombre de cycles des méta-répétitions. Ces indicateurs de dispersion sont l'écart-type, le rapport (médiane-minimum)/minimum et le rapport (maximum-médiane)/maximum.

Différentes courbes représentant ces indicateurs de dispersion en fonction du nombre de warmups ont été tracées afin de fixer le nombre de warmups pour la suite du projet.

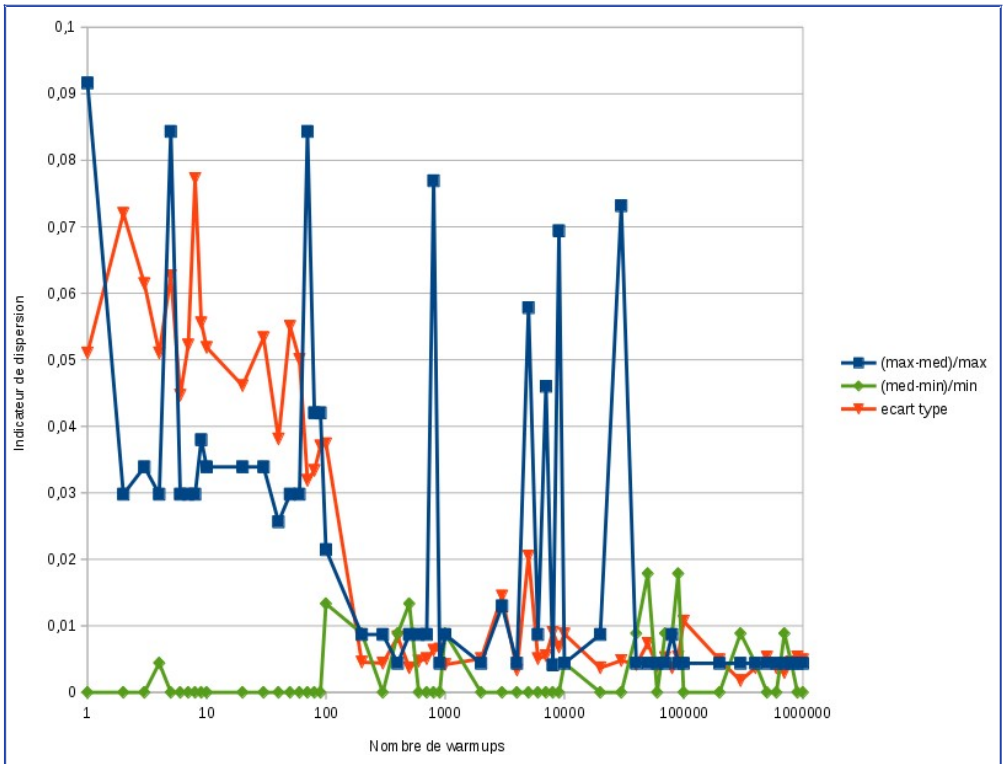


Illustration 8: Courbes des indicateurs de dispersion en fonction du nombre de warmups

On peut voir sur ces courbes qu'en choisissant un nombre de warmups supérieur à 100000, les indicateurs de dispersion calculés sont tous inférieurs à 1%. Nous allons donc fixer le nombre de warmups pour l'étude du cache L3 à 200000.

4.2 Mesures

4.2.1 Options de compilation

Nous allons comparer les moyennes du nombre de cycles par itération avec différentes options de compilation. Nous allons utiliser le compilateur gcc avec les options -o2, -o3, -o3 -march=native, -ofast, -ofast -march=native, ainsi que le compilateur icc avec les options -o2, -o3 et -o3 -xHost.

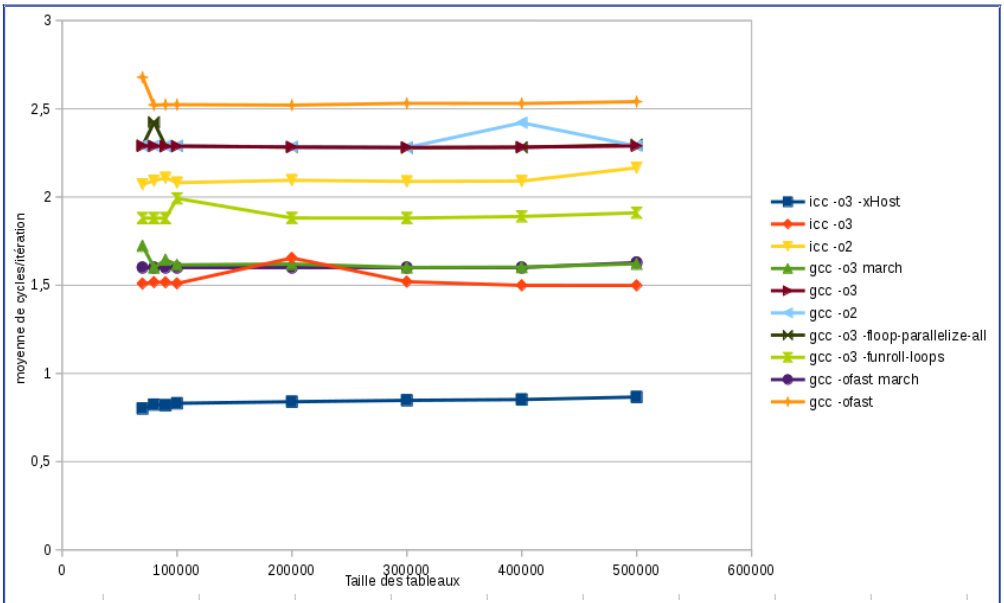


Illustration 9: Courbes des moyennes de cycles en fonction de la taille des tableaux

Nous pouvons tout d'abord observer que l'option de compilation Ofast avec gcc augmente le nombre de cycles/itération en moyenne par rapport à O2.
La compilation avec icc et l'option -O3 -xHost au contraire semble donner l'exécutable donnant le moins de cycles/itérations.

Pour étudier les speed-up des différents compilateurs, nous allons utiliser la moyenne pour la taille des tableaux à 200000, la valeur de référence étant gcc -O2 (speed-up de gcc -O2 à 1).

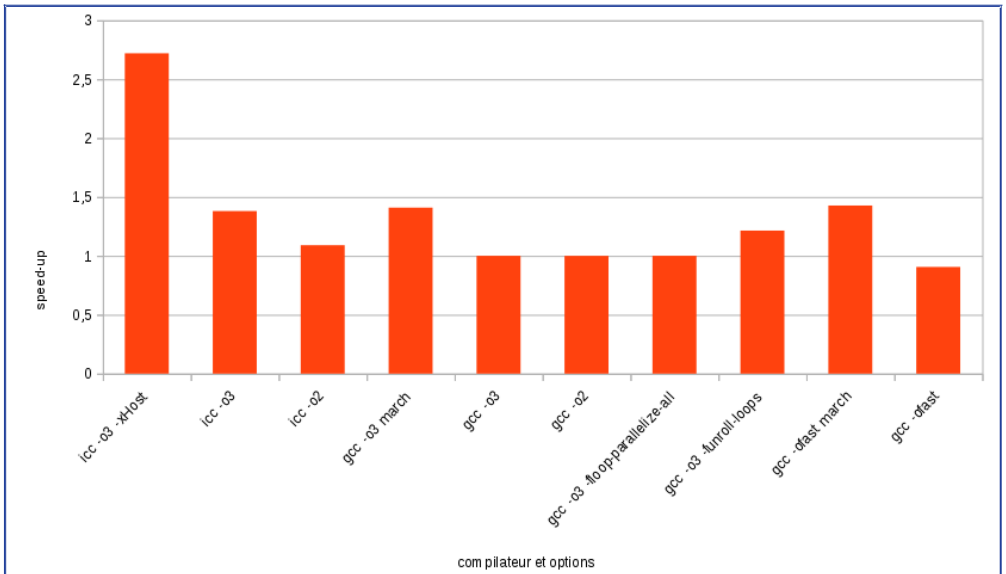


Illustration 10: HIstogramme des moyennes de cycles pour une taille de 200000

Nous avons donc un speed-up de plus de 2,7 pour icc -O3 -march=native, de loin le meilleur speed-up de l'échantillon. Nous avons ensuite icc -O3, gcc -O3 -march=native et icc -Ofast -march=native ayant un speed-up proche de 1,4.

La compilation avec icc -O2 produit un speed-up près de 1,1 et gcc -O3 -funrool-loops un speed-up d'environ 1,2.

gcc -O3 et gcc -O3 -floop-parallelize-all quant à eux donnent un speed-up très légèrement supérieur à 1 (à 10^{-4} près).

Comme observé précédemment, le speed-up pour gcc -Ofast est inférieur à 1, il est de 0,9.

4.2.2 Étude des exécutables avec Maqao

4.2.2.1 Compilateur GCC

Nous avons débuté l'analyse avec Maqao en utilisant un exécutable produit avec une compilation gcc -O2. La complexité du flux est mauvaise, et Maqao recommande l'utilisation de l'option de compilation -funroll-loops. Le programme passe 99,78 % du temps dans des boucles, qui pour la partie kernel ne sont pas du tout vectorisées dans le fichier kernel.c. Maqao annonce un speed-up de 7,88 pour une vectorisation totale. Le programme prend 36,94 secondes à s'exécuter. Tous les speed-ups seront calculés basé sur ce temps d'exécution.

Global Metrics

Total Time (s)		36.94
Time in loops (%)		99.78
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.88
	Nb Loops to get 80%	2

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 6	6-13	go2:kernel.c	baseline	52.5	19.39	0
Loop 5	6-19	go2:kernel.c	baseline	47.22	17.44	0
Loop 3	13-14	go2:driver.c	main	0.03	0.01	25
Loop 4	13-14	go2:driver.c	main	0.03	0.01	25
Loop 0	51-52	go2:driver.c	main	0	0	0

Illustration 11: Métriques et résumé des boucles fourni par Maqao pour gcc -O2

Nous avons ensuite étudié le résultat de la compilation avec gcc -O3. Là encore, Maqao recommande l'option -funroll-loops et la complexité du flux est de 2. La vectorisation des boucles est encore de 0 % pour les boucles de kernel.c, et le programme aura les mêmes métriques qu'avec -O2, sauf le temps d'exécution qui est légèrement meilleur à 36,75s, soit un speed-up de

Global Metrics

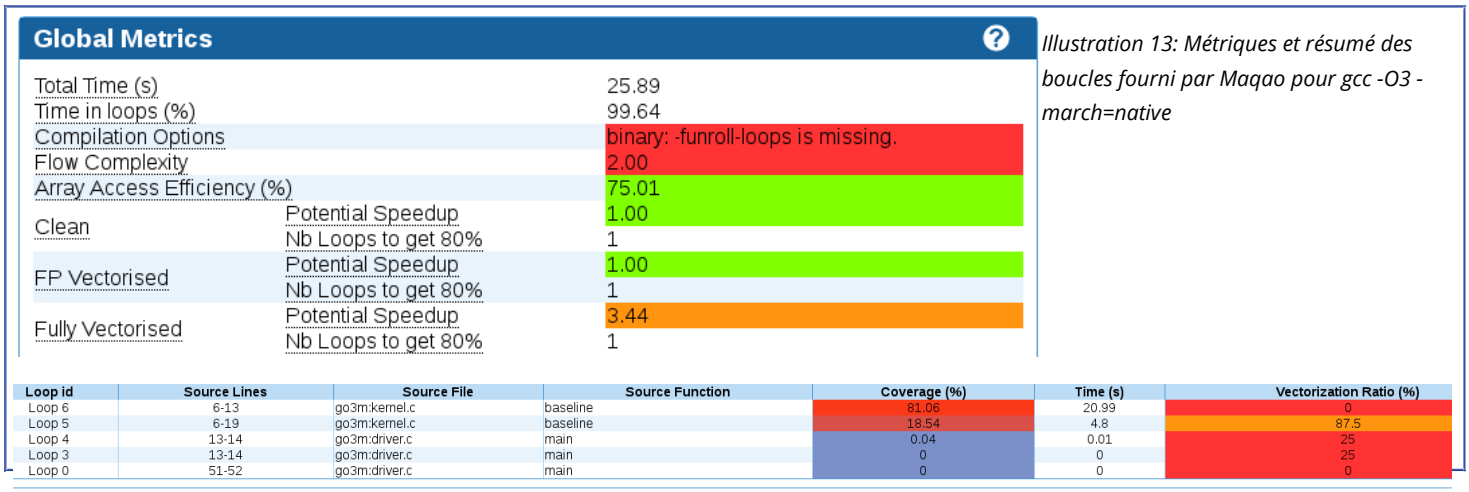
Total Time (s)		36.75
Time in loops (%)		99.78
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.88
	Nb Loops to get 80%	2

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 6	6-13	go3:kernel.c	baseline	52.08	19.14	0
Loop 5	6-19	go3:kernel.c	baseline	47.64	17.51	0
Loop 3	13-14	go3:driver.c	main	0.03	0.01	25
Loop 4	13-14	go3:driver.c	main	0.03	0.01	25
Loop 0	51-52	go3:driver.c	main	0	0	0

Illustration 12: Métriques et résumé des boucles fourni par Maqao pour gcc -O3

1,005.

Le prochain exécutable analysé est le produit de gcc -O3 -march=native. Ici on peut noter certaines différences par rapport aux options précédentes, notamment qu'une boucle de kernel.c est vectorisée à 87,5 %. Le temps passé dans les boucles est ici de 99,64 %, et le speed-up avec vectorisation total est de 3,44. Nous avons un temps d'exécution de 25,89s, soit un speed-up de 1,45.



Global Metrics

Total Time (s)		25.89
Time in loops (%)		99.64
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.44
	Nb Loops to get 80%	1

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 6	6-13	go3m:kernel.c	baseline	81.06	20.99	0
Loop 5	6-19	go3m:kernel.c	baseline	18.54	4.8	87.5
Loop 4	13-14	go3m:driver.c	main	0.04	0.01	25
Loop 3	13-14	go3m:driver.c	main	0	0	25
Loop 0	51-52	go3m:driver.c	main	0	0	0

Illustration 13: Métriques et résumé des boucles fourni par Maqao pour gcc -O3 -march=native

Ensuite, nous avons ajouté l'option -funroll-loops. La vectorisation qui était à 87,5 est passée à 93,75. Maqao nous indique que le speed-up sera à 1 si complètement vectorisé, donc vectoriser plus n'apportera rien. En revanche, l'efficacité d'accès aux tableaux est passée de 75,01 à 40,67. De plus le speed-up potentiel obtenu en nettoyant les instructions effectuant des calculs d'adresses et d'entiers scalaires dans les boucles analysées est de 1,05. Il était jusqu'à présent toujours à 1. La complexité du flux à aussi augmenté, elle est passée à 3,99. Le programme met 22,22s à s'exécuter soit un speed-up de 1,66.

Global Metrics

Total Time (s)		22.22
Time in loops (%)		99.61
Compilation Options		OK
Flow Complexity		3.99
Array Access Efficiency (%)		40.67
Clean	Potential Speedup	1.05
	Nb Loops to get 80%	5
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 6	6-13	go3mu:kernel.c	baseline	78.22	17.38	0
Loop 5	19-19	go3mu:kernel.c	baseline	21.29	4.73	93.75
Loop 3	13-14	go3mu:driver.c	main	0.05	0.01	20.51
Loop 4	13-14	go3mu:driver.c	main	0.05	0.01	20.51
Loop 0	51-52	go3mu:driver.c	main	0	0	0

Illustration 14: Métriques et résumé des boucles fourni par Maqao pour gcc -O3 -march=native -funroll-loops

4.2.2.2 Compilateur ICC

Nous avons ensuite étudié la compilation avec icc. Tout d'abord, avec l'option -O2, nous observons que l'on nous recommande l'option -Xhost ou -xCORE. De plus, l'une des deux boucles de kernel.c a été complètement vectorisée. La complexité du flux est de 2,71, et Maqao nous informe qu'en vectorisant complètement 2 boucles, on peut obtenir un speed-up de 5,54. En comparant avec gcc -O2, on observe que ici on passe beaucoup plus de temps dans la première des boucles de kernel.c (ici notée loop 5, loop 6 pour gcc -O2), mais beaucoup moins dans la deuxième, celle vectorisée.
Le speed-up est médiocre, avec un temps d'exécution de 34,14 on a 1,08.

Global Metrics

Total Time (s)		34.14
Time in loops (%)		99.75
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		2.71
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.40
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	5.54
	Nb Loops to get 80%	2

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 5	6-13	io2:kernel.c	baseline	85.43	29.17	0
Loop 7	16-19	io2:kernel.c	baseline	14.26	4.87	100
Loop 3	13-14	io2:driver.c	main	0.03	0.01	25
Loop 4	13-14	io2:driver.c	main	0.03	0.01	25
Loop 2	51-52	io2:driver.c	main	0	0	0

Illustration 15: Métriques et résumé des boucles fourni par Maqao pour icc -O2

Pour icc -O3, Maqao préconise encore l'utilisation de -Xhost ou -xCORE. On a une complexité de

flux de 1, bien inférieure à ce que l'on avait avant. Les deux boucles de kernel.c ont été vectorisées à 100 %, et on observe que presque toutes les métriques sont les meilleures obtenues avec gcc -O3. Nous obtenons un speed-up de 1,53 par rapport à gcc -O2.

Global Metrics

Total Time (s)		24.19
Time in loops (%)		99.64
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.24
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.99
	Nb Loops to get 80%	2

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 10	3-13	io3:kernel.c	baseline	79.26	19.17	100
Loop 6	16-19	io3:kernel.c	baseline	20.34	4.92	100
Loop 3	13-14	io3:driver.c	main	0.04	0.01	25
Loop 4	13-14	io3:driver.c	main	0	0	25

Illustration 16: Métriques et résumé des boucles fourni par Maqao pour icc -O3

Enfin, nous avons compilé avec icc -O3 -xHost. Nous avions observé précédemment que les résultats étaient bien meilleurs ainsi, et Maqao nous le confirme. En effet, tous les speed-up potentiels indiqué par Maqao sont à 1, la complexité de flux aussi. L'efficacité d'accès aux tableaux à 75,02 est même très légèrement supérieure à ce que l'on avait avant (75,01 pour toutes les compilations sauf avec gcc -O3 -funroll-loops -march=native où l'on avait 40,67). Nous pouvons aussi noter une différence avec les boucles, seulement ici nous n'avons pas que deux boucles pour kernel.c et deux boucles pour driver.c, nous avons en plus duex autres boucles de kernel.c et une de baseline. Toutes les boucles de kernel.c à l'exception de la dernière ont été vectorisées à 100 %.

Nous avons ici le meilleur speed-up, à 2,68 avec 13,76 secondes d'exécution.

Global Metrics

Total Time (s)		13.76
Time in loops (%)		99.42
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.02
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)
Loop 10	6-13	io3x:kernel.c	baseline	68.61	9.44	100
Loop 7	16-19	io3x:kernel.c	baseline	30.67	4.22	100
Loop 4	13-14	io3x:driver.c	main	0.07	0.01	25
Loop 3	13-14	io3x:driver.c	main	0.07	0.01	25
Loop 6	16-19	io3x:kernel.c	baseline	0	0	100
Loop 8	16-19	io3x:kernel.c	baseline	0	0	0
Loop 2	51-52	io3x:driver.c	main	0	0	0

Illustration 17: Métriques et résumé des boucles fourni par Maqao pour icc -O3-xHost

5 Travaux sur la RAM – Pierre VERDURE

N'ayant pas de machine sous Linux nativement, et la VM offrant des résultats aléatoires, les mesures ont été faites sur la machine de Clément, qu'il m'a aimablement prêté.

5.1 Configuration

5.1.1 Système

Les tests ont été effectués sous Ubuntu 18.14.1 LTS, installé en natif (via un dual-boot) sur un Macbook Pro mi-2012.
Pour plus de fiabilité, les tests ont été réalisés avec la machine branchée sur secteur, et le turbo-boost désactivé.

Processeur	Intel Core i5 3210M
Nombre de cœurs	2
Nombre de threads	4
Fréquence	2.5 Ghz
Taille du cache L1	32 Ko (par cœur)
Taille du cache L2	256 Ko (par cœur)
Taille du cache L3	3 Mo
Taille de la RAM	16 Go

Tableau 4: Informations du processeur

Illustration 18: Représentation du processeur (via likwid-topology)

5.1.2 Taille de données manipulées

On souhaite travailler sur la RAM, de taille 16 Go.

Afin de s'assurer que nous travaillons bien dans la RAM, et non pas dans un cache, il est préférable de travailler avec des données de taille 3 fois supérieure à la taille du cache L3 (ici 3 Mo).

Il n'est pas nécessaire d'utiliser une majeure partie de la RAM, 100 Mo sont largement suffisants.

On a donc :

$$9\,000\,000 < T_{RAM} < 100\,000\,000$$

Or, dans notre sujet, nous manipulons deux tableaux (de même taille, notée n) à une dimension de flottants.

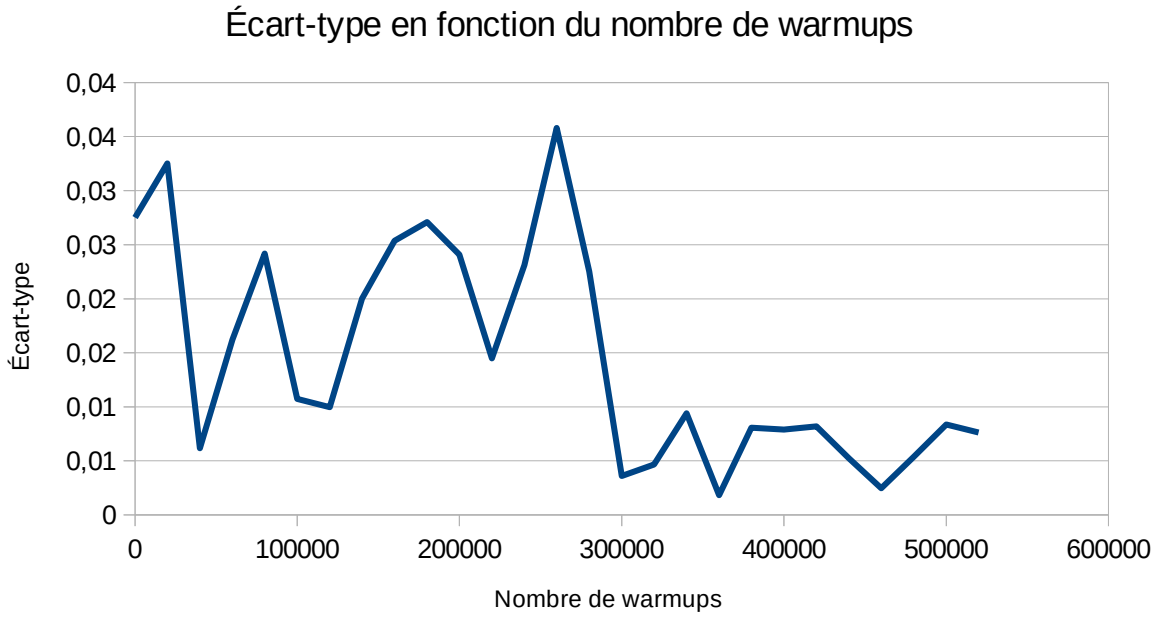
Un flottant étant codé sur 4 octets, il vient :

$$(9\,000\,000 < 2 * n * 4 < 100\,000\,000) \Leftrightarrow 1\,125\,000 < n < 12\,500\,000$$

Pour la suite du sujet, on prendra donc **n = 2500000**.

5.1.3 Calcul du nombre de warmups

La taille de données étant conséquente, le temps d'exécution l'est également, je n'ai donc pas pu réaliser l'ensemble des indicateurs pour calculer le nombre de répétitions warmups, j'ai dû me contenter du principal, l'écart-type.



On remarque que l'écart-type converge dès 300 000 warmups, nous prendrons donc cette valeur pour la suite des mesures.

5.2 Mesures

5.2.1 GCC -O2

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	4,32258064516129
Médiane (cycles/itérations)	4,32

MAQAO souhaite que l'on utilise le flag -funroll-loops afin d'autoriser le déroulage de boucles.

Autre point négatif, la complexité des flux est élevée.

On remarque que, si le code était totalement vectorisé, on pourrait atteindre un speed-up de 5.8.

Global Metrics

Total Time (s)		54.62
Time in loops (%)		96.6
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.99
Array Access Efficiency (%)		75.13
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	5
Fully Vectorised	Potential Speedup	5.80
	Nb Loops to get 80%	2

On remarque que le temps de passé dans les fonctions est équivalent dans les deux boucles :

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 6	4-10	baseline:kernel.c	baseline	48.81
Loop 5	13-15	baseline:kernel.c	baseline	46.76
Loop 3	13-14	baseline:driver.c	main	0.55
Loop 4	13-14	baseline:driver.c	main	0.48

5.2.2 GCC -O3

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	3,79709677419355
Médiane (cycles/itérations)	3,79

Global Metrics

Total Time (s)		47.42
Time in loops (%)		95.7
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.56
Array Access Efficiency (%)		75.12
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.09
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	2.52
	Nb Loops to get 80%	2

Même remarque que pour GCC -O2 concernant le flag -funroll-loops.

Cependant, on remarque que, du fait des optimisations supplémentaires apportées par -O3, la complexité du flux diminue.

Le programme est donc plus performant, le speed-up potentiel, en mettant en place une vectorisation totale n'est plus que 2.52.

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 7	13-15	baseline:kernel.c	baseline	53.82
Loop 8	10-10	baseline:kernel.c	baseline	21.17
Loop 9	4-8	baseline:kernel.c	baseline	19.82
Loop 3	13-14	baseline:driver.c	main	0.55
Loop 4	13-14	baseline:driver.c	main	0.34

On remarque que l'une des deux boucles est divisée en deux, afin d'être vectorisée.

5.2.3 GCC -O3 -march=native

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	2,46741935483871
Médiane (cycles/itérations)	2,5

On observe encore une amélioration des performances, par rapport aux degrés d'optimisations -O2 et -O3.

Global Metrics

Total Time (s)		29.64
Time in loops (%)		93.18
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.30
Array Access Efficiency (%)		75.20
Clean	Potential Speedup	1.40
	Nb Loops to get 80%	6
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	6
Fully Vectorised	Potential Speedup	1.13
	Nb Loops to get 80%	2

Ici encore, on observe une amélioration des performances du programme.

La complexité du flux n'est plus que de 1.3.

De plus, le speed-up potentiel n'est plus que de 1.13.

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 8	10-10	baseline:kernel.c	baseline	32.59
Loop 9	4-8	baseline:kernel.c	baseline	31.58
Loop 7	13-15	baseline:kernel.c	baseline	27.53
Loop 4	13-14	baseline:driver.c	main	0.74
Loop 3	13-14	baseline:driver.c	main	0.74

La gestion des boucles reste identique à -O3, une des deux boucles est scindée en deux.
Le temps passé dans les trois différentes boucles est assez uniforme (environ 30% dans chacune d'elles).

5.2.4 ICC -O2

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	4,16096774193548
Médiane (cycles/itérations)	4,16

Les performances relevées sont sensiblement meilleures que celles de -O2.

Global Metrics

Total Time (s)		50.78
Time in loops (%)		95.97
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		2.50
Array Access Efficiency (%)		75.12
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.58
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.14
	Nb Loops to get 80%	2

MAQAO demande l'ajout de l'option de compilation -xHost, qui met en place des optimisations spécifiques au CPU.

La complexité de flux est supérieure à celle relevée en -O2.

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 5	4-10	baseline:kernel.c	baseline	72.15
Loop 7	13-15	baseline:kernel.c	baseline	22.92
Loop 4	13-14	baseline:driver.c	main	0.47
Loop 3	13-14	baseline:driver.c	main	0.43

Au niveau des boucles, à l'instar de GCC -O2, aucune modification n'est apportée au code.

5.2.5 ICC -O3

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	4,38612903225806
Médiane (cycles/itérations)	4,38

Étrangement, ce degré d'optimisation nuit aux performances du programme, puisque le nombre de cycles par itérations est supérieur à celui relevé en ICC -O2.

Global Metrics

Total Time (s)		54.4
Time in loops (%)		96.29
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.11
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.24
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.36
	Nb Loops to get 80%	2

Pourtant, MAQAO nous indique que la complexité du flux est optimale (1.00).

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 10	2-10	baseline:kernel.c	baseline	74.82
Loop 6	13-15	baseline:kernel.c	baseline	20.62
Loop 4	13-14	baseline:driver.c	main	0.48
Loop 3	13-14	baseline:driver.c	main	0.37

En analysant la première boucle, MAQAO nous dit qu'elle est déroulée par 4, mais que la vectorisation n'est pas optimale, puisque seule 62 % de la taille du registre est occupée.
On pourrait donc obtenir un speed-up de 1.33 au niveau de cette boucle.

5.2.6 ICC -O3 -xHost

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	2,85258064516129
Médiane (cycles/itérations)	2,83

Grâce au flag -xHost, ICC va mettre en place des optimisations propres au CPU. Ceci explique que le programme généré est plus performant que celui généré par ICC -O2 et ICC -O3.

Global Metrics

Total Time (s)		35.24
Time in loops (%)		93.64
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		80.39
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	1

Les métriques indiquées par MAQAO sont toutes au vert, ce qui signifie que le programme généré est relativement performant.

En effet, en vectorisant totalement le code, on pourrait obtenir un speed-up potentiel de seulement 4 %, ce qui est relativement peu.

Loops Index

Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 10	4-10	baseline:kernel.c	baseline	59.25
Loop 6	13-15	baseline:kernel.c	baseline	33.54
Loop 4	13-14	baseline:driver.c	main	0.57
Loop 3	13-14	baseline:driver.c	main	0.28

MAQAO nous indique que les boucles ont été déroulées par 16 ici, contre seulement par 4 précédemment.

5.2.7 GCC -Ofast

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	3,80548387096774
Médiane (cycles/itérations)	3,79

Les résultats sont semblables à ceux relevés par GCC -O3.
Les optimisations ajoutées par -Ofast ne font que peu de différences, et peuvent nuire au fonctionnement du programme.

5.2.8 GCC -Ofast -march=native

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	2,41096774193548
Médiane (cycles/itérations)	2,41

Ici encore, les résultats observés sont semblables à ceux relevés par GCC -O3 -march=native.

5.2.9 GCC -O3 -funroll-loops -march=native

Pour n = 2 500 000 :

Moyenne (cycles/itérations)	2,48322580645161
Médiane (cycles/itérations)	2,48

Global Metrics

Total Time (s)		29.66
Time in loops (%)		91.11
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.42
Clean	Potential Speedup	1.01
	Nb Loops to get 80%	5
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	5
Fully Vectorised	Potential Speedup	1.11
	Nb Loops to get 80%	1

Loops Index

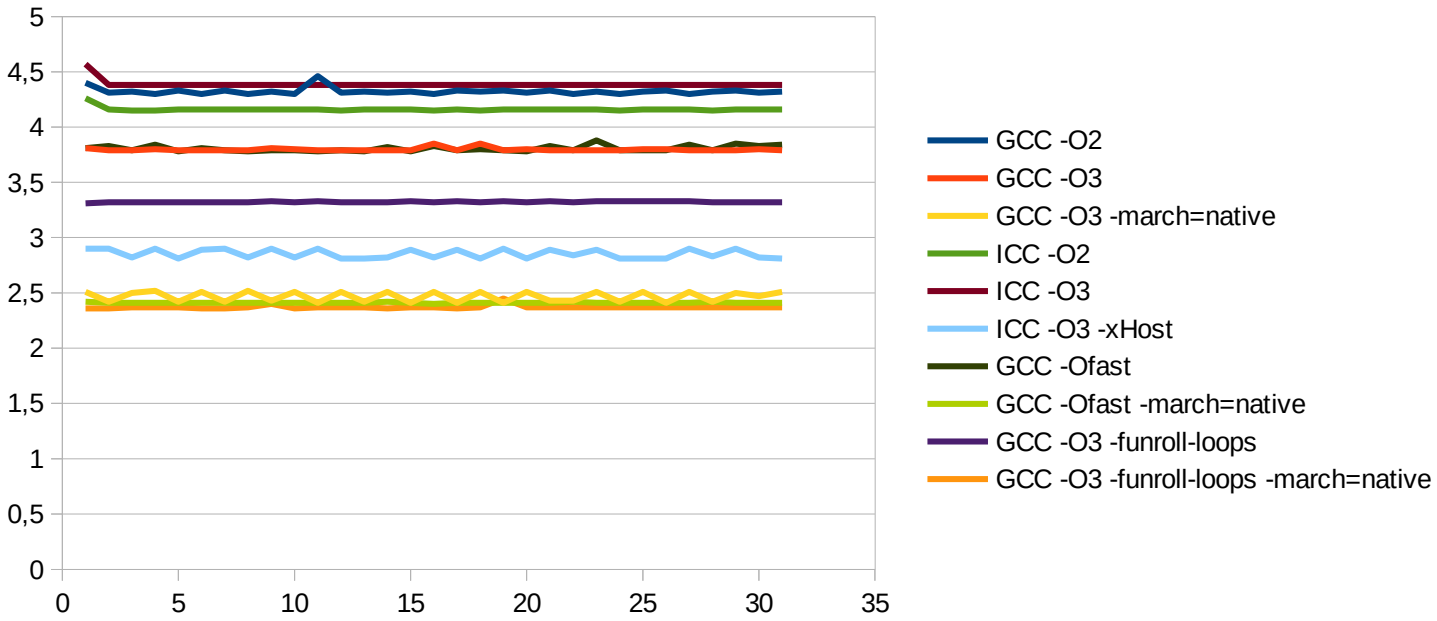
Coverage (%) | Time (s) | Vectorization Ratio (%) | Speedup If Clean | Speedup If FP Vectorized | Speedup If Fully Vectorized | Select all

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Vectorization Ratio (%)
Loop 7	10-10	baseline:kernel.c	baseline	32.1	100
Loop 9	4-8	baseline:kernel.c	baseline	31.09	100
Loop 5	15-15	baseline:kernel.c	baseline	26.3	0
Loop 4	13-14	baseline:driver.c	main	0.88	25
Loop 3	13-14	baseline:driver.c	main	0.74	25

Deux des trois boucles sont vectorisées, la dernière ne peut l'être, notamment à cause des expressions conditionnelles.

5.3 Conclusion

Afin de comparer les différentes optimisations mises en place au sein de ce rapport, on peut tracer la courbe suivante :



On remarque que les trois optimisations les plus performantes sont :

- gcc -O3 -march=native
- gcc -Ofast -march=native
- gcc -O3 -funroll-loops -march=native

Les optimisations mises en places par le flag -march=native jouent donc un rôle primordial pour la performance du programme.

Webographie

https://gcc.gnu.org/onlinedocs/

https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference

https://wiki.gentoo.org/wiki/GCC_optimization/fr

https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr