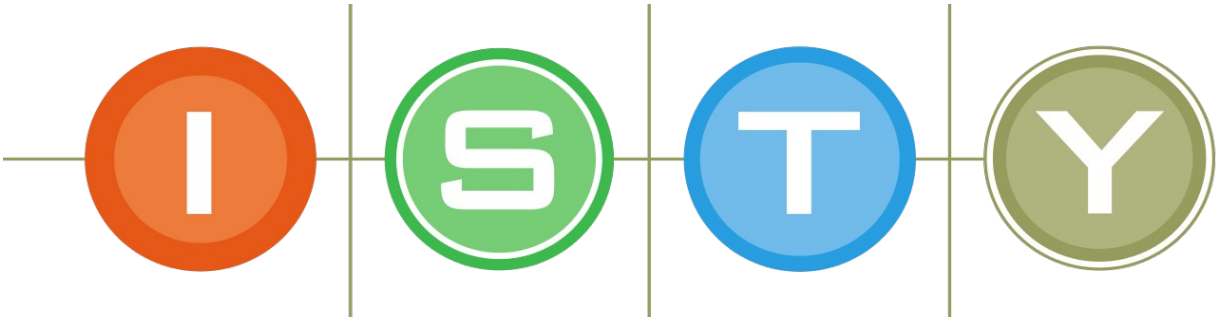


Architecture des Ordinateurs Avancée

Phase II – 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 seconde phase du projet d'Architecture des Ordinateurs Avancées (AOA), unité d'enseignement du second semestre de IATIC4.

Cette phase fait suite à une première phase, au sein de laquelle nous avons testé et mis en place différentes optimisations lors de la compilation.

Ici, on s'intéressera donc principalement aux optimisations liées au code-sources (optimisation de boucles, parallélisation avec OpenMP, etc.)

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 différentes versions du noyau que nous avons mis en place.
Ensuite, nous détaillerons les travaux effectués sur les différents niveaux de caches, c'est à dire les différentes mesures réalisées, en expliquant les résultats obtenus.

1 Travaux communs

1.1 Script de mesure (partie 1)

Nous avons mis en place un nouveau script afin de nous permettre d'effectuer les mesures. Il s'agît du fichier phase2.sh.

Contrairement aux scripts de la phase 1, celui-ci nous permet de réaliser en même temps les différentes mesures à effectuer en faisant varier la taille des tableaux, mais lance également la commande associée à l'utilitaire MAQAO.

De plus, le script met en place une arborescence plus claire et organisée que celle mise en place lors de la phase 1.

Cette arborescence se présente de la manière suivante : un dossier *measure* contient plusieurs sous-dossiers, chacun associé à une optimisation (par exemple *NO* pour les mesures de la version originale, *OPT1* pour les mesures de la première optimisation, etc.).

Au sein de ces dossiers, on retrouve à chaque fois un fichier *.csv,* contenant les mesures effectuées en faisant varier la taille des tableaux, et qui sera exploité par la suite pour tracer les courbes, ainsi qu'un dossier MAQAO contenant le rapport fournit par l'utilitaire.

Une fois l'arborescence crée, on exécute MAQAO puis on effectue les mesures en faisant varier les tailles des tableaux de la manière suivante :

```
# MAQAO
echo "MAQAO"
echo "OPT $1"
sudo maqao oneview --create-report=one binary=baseline xp=measures/$OPT/MAQAO
run_command="<binary> $TAILLE_TABS $NB_WARMUPS_MAQAO $NB_REPETS $VAL"
echo "DONE"

# VAR MEASURE
echo "VAR MEASURE"
FILE=measures/$OPT/$OPT.csv
```

```
echo "Taille tableaux" > $FILE
for TAILLE_TABS in `seq $BORNE_INF_TABS $STEP_TABS $BORNE_SUP_TABS`;
do
    echo "Taille : $TAILLE_TABS"
    printf "$TAILLE_TABS;" >> $FILE
    taskset -c 0 ./baseline $TAILLE_TABS $NB_WARMUPS $NB_REPETS $VAL | sed
's/[^0-9\.]*//g' | tr '\n' ';' >> $FILE
    echo >> $FILE
done
echo "DONE"
```

Utiliser ce script nous a donc apporté un gain de temps relativement conséquent.

1.2 Script de mesure (partie 2)

Le premier script permettant de tester les différentes optimisations apportées au niveau du code source, nous avons souhaité mettre en place un second script (appelé *phase2pt2.sh*), permettant lui de tester les différents degrés de compilation.

Nous avons donc mis en place une arborescence semblable à celle du premier script, composée de sous-dossiers nommés selon le degré de compilation (typiquement *gcc_-O3-march=native*).

Comme nous avions déjà connaissance des degrés d'optimisations les plus intéressants/agressifs (déterminés lors de la phase 1), nous avons procédé de la manière suivante, pour chaque degré d'optimisation :

1.Compilation, de la manière suivante :

```
CC=gcc
CFLAGS="-O3 -march=native -g -Wall"
echo "COMPILATEUR = $CC $CFLAGS"
make clean
make all CC="$CC" CFLAGS="$CFLAGS" OPT=OPT3 # On prend la dernière option
```

Nous redéfinissons donc à chaque fois le compilateur et les différents flags associés, et on passe les variables au Makefile.

Petite particularité et limite de cette méthode, pour que cela ait un effet, il est nécessaire de commenter les deux premières lignes du fichier Makefile, où sont justement définies les variables CC et CFLAGS.
A noter que nous aurions pu pallier à cela en modifiant directement le contenu du Makefile via le script, à l'aide de *sed* par exemple.

2. Exécution et mesures : de la même manière que pour le premier script.

1.3 Script de mesure OpenMP

Pour la partie OpenMP de cette phase du projet, nous avons souhaité mettre en place qui exécuterait le programme pour un nombre de threads variant (*omp.h*).
En effet, le nombre de threads le choix du nombre de threads a un impact primordial sur l'exécution d'un programme parallélisé, et cette valeur est d'ailleurs propre à chaque machine.

La compilation est effectuée une seule fois (le code ne change pas, on agît uniquement sur le nombre de threads. On notera toutefois la présence (et nécessité) du flag *-fopenmp*.

```
# COMPILATION
CC=gcc
CFLAGS="-O2 -fopenmp -g -Wall"
make clean
make all CC="$CC" CFLAGS="$CFLAGS" OPT=OMP # On prend la version parallélisée
```

Ensuite, on fait varier le nombre de threads de la manière suivante :

```
# Exécution pour un nombre de threads variant entre 1 et 5
for OMP_NUM_THREADS in `seq 1 1 5`;
do
    echo "$OMP_NUM_THREADS THREADS"
    export OMP_NUM_THREADS
    [… MAQAO & Mesures …]
done
```

A noter qu'il est nécessaire d'exporter la valeur OMP_NUM_THREADS, qui est une variable d'environnement propre à OpenMP.

2 Travaux sur le cache L1 - Lois JULLIEN

2.1 Optimisations du code source

Nous allons exécuter différentes optimisations de codes pour enfin selectionner la meilleure grâce à Maqao. Nous prenons des tableaux de taille entre 2000 et 3800 correspondant au cache L1.

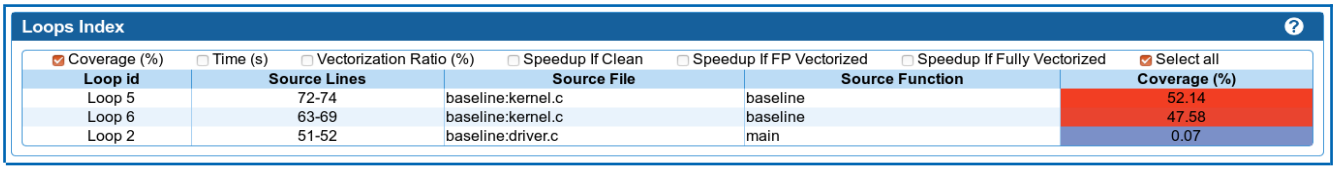
2.1.1 Version originale

```
/* Original */
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;
        }
}

#endif
```

Voici la version originale du code, nous exécutons Maqao afin d'avoir une base sur les différentes optimisations à venir.

Global Metrics

Total Time (s)		28.08
Time in loops (%)		99.79
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.00
Array Access Efficiency (%)		75.02
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	7.88
	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	72-74	baseline:kernel.c	baseline	52.14
Loop 6	63-69	baseline:kernel.c	baseline	47.58
Loop 2	51-52	baseline:driver.c	main	0.07

La version originale exécute le programme en un temps total de 28 secondes. L'option Funroll-loops est manquant (nous allons par la suite l'ajouter car nous exécutons avec O2 pour l'instant). La complexité de flux est trop élevée.
Maqao nous indique aussi les optimisations possible sur une des boucle, nous choisissons la première. Cette boucle peut être déroulée (grâce aux options de compilation) et nous pouvons enlever les expressions conditionnelles en les plaçant par exemple en dehors de la boucle.

Coverage	52.14 %
Function	baseline
Source file and lines	kernel.c:72-74
Module	baseline

The loop is defined in /home/lois/Desktop/projet_aob/kernel.c:72-74.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

2.1.2 Optimisation 1

L'optimisation numéro 1 consiste à sortir au maximum les conditions des boucles (comme nous indiquait maqao pour le programme basique). Nous pouvons évidemment pas sortir toutes les conditions mais nous pouvons split la première boucle en 2 afin d'enlever un else if du programme basique. Cela donne :

```
#ifdef OPT1
/* 1ère optimisation - If Hoisting */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                if (a[i] > x)
                        b[i] = a[i];
                else
                        b[i] = x;
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
        }

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

Exécutons maintenant Maqao sur ce nouveau programme :

Global Metrics

Total Time (s)		25.6
Time in loops (%)		99.54
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.62
Array Access Efficiency (%)		75.10
Clean	Potential Speedup	1.12
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.75
	Nb Loops to get 80%	3

Nous pouvons dans un premier temps remarquer que nous gagnons 3 secondes sur le temps d'exécution, cela nous permet déjà d'affirmer que cette optimisation est utile.
L'option de compilation est toujours manquante et nous pouvons encore optimiser.

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	16-18	baseline:kernel.c	baseline	62.19
Loop 7	5-9	baseline:kernel.c	baseline	18.52
Loop 6	12-13	baseline:kernel.c	baseline	18.44
Loop 2	51-52	baseline:driver.c	main	0.39

Nous avons une boucle en plus, développons par exemple la première afin de voir se que nous informe Maqao.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

Maqao nous informe que nous pouvons encore sortir des conditions de la boucle pour gagner en optimisation, nous l'avons fait pour cette optimisation mais nous pouvons pas le faire plus.
Cette optimisation fonctionne, nous allons donc la garder pour l'optimisation numéro 2.

2.1.3 Optimisation 2

L'optimisation numéro 2 consiste (en plus de l'optimisation 1) de procéder à un loop-fusion.

```
/* 2ème optimisation - If Hoisting & Loop fusion */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                if (a[i] > x)
                        b[i] = a[i];
                else
                        b[i] = x;

                if(b[i] < 0.0)
                        b[i] = 0.0;
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;

                if(b[i] < 0.0)
                        b[i] = 0.0;
        }
}
```

Exécutons maintenant Maqao.

Global Metrics

Total Time (s)		18.24
Time in loops (%)		100
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.88
Array Access Efficiency (%)		71.08
Clean	Potential Speedup	1.12
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.19
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.60
	Nb Loops to get 80%	2

Nous sommes passé de 25 sec de temps d'exécution à 18, nous gagnons donc 8 secondes mais le flow complexity reste trop elevé.

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	26-33	baseline:kernel.c	baseline	62.72
Loop 5	36-40	baseline:kernel.c	baseline	37.28

Nous avons donc maintenant seulement deux boucles.

2.1.4 Optimisation 3

L'optimisation 3 consiste à remplacer les expressions conditionnelles.

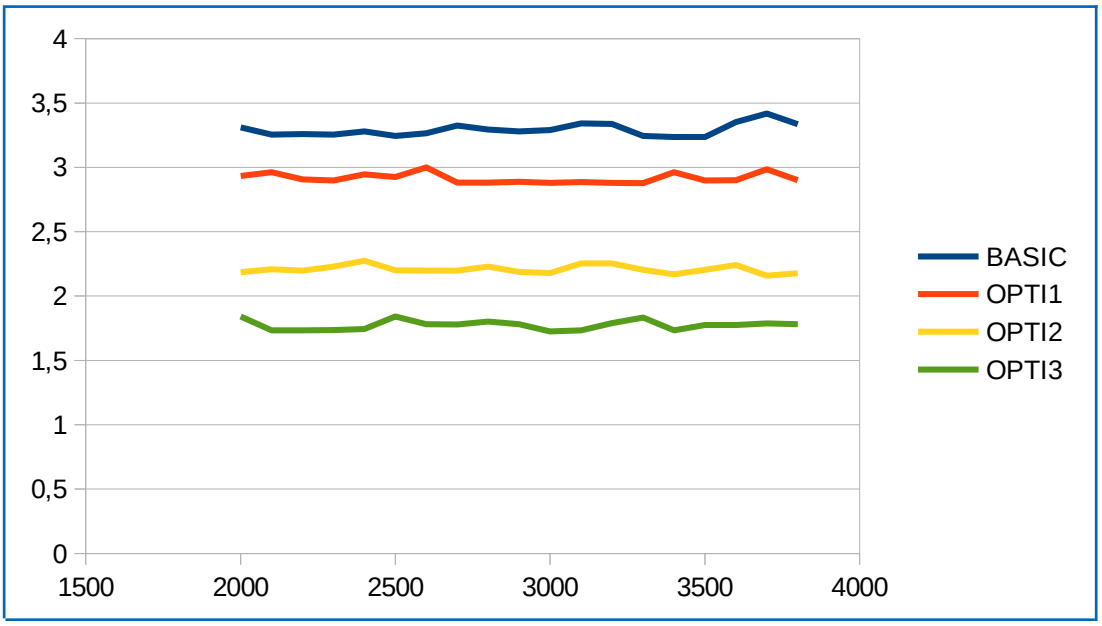
```
/* 3ème optimisation - Remplacement des expressions conditionnelles */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                b[i] = (a[i] > x ? a[i] : x);
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }
}
```

Exécutons maintenant Maqao.

Global Metrics

Total Time (s)		14.54
Time in loops (%)		99.87
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.04
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.31
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.93
	Nb Loops to get 80%	2

Nous gagnons encore 4 secondes sur l'exécution, le flow-complexity est maintenant à 1 et le potentiel de speedup sur le code en lui même est à 1. Avant de finir notre comparaison, nous allons tracer un graphique représentant la version de base avec les 3 optimisations. Le graphe représente la moyenne des cycles par itérations (temps) en fonction de la taille des tableaux :



Ce résultat nous confirme se que l'on disait avant, à chaque fois les optimisations nous font gagner en temps. Nous garderons bien l'optimisation 3 qui regroupe les 3 optimisations.

2.2 Utilisation d'optimisations plus agressives

On va maintenant modifier la compilations de base (GCC -O2) avec notre code source optimisé (l'optimisation 3).

Au sein de la première phase du projet, nous avions pu remarquer que l'optimisation gcc -O3 -march=native -funroll-loops, avec le code-source de base, produisait le meilleur résultat en termes cycles par itérations.
On va considérer ici uniquement les 4 optimisations les plus efficaces lors de la phase 1 sur cette machine, à savoir, du moins efficace au plus efficace :

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

2.2.1 ICC -O3 -xHost

En compilant avec icc -O3 -xHost, on obtient les résultats suivants :

Taille tableaux	MOY ICC	MED ICC	ECARTYPE
2000	1,07	1,07	0
2100	1,07	1,07	0
2200	1,078065	1,07	0,03004656
2300	1,071613	1,07	0,00898027
2400	1,07	1,07	0
2500	1,07	1,07	0
2600	1,071613	1,07	0,00898027
2700	1,07129	1,07	0,00718421
2800	1,071613	1,07	0,00898027
2900	1,063871	1,06	0,00495138
3000	1,079032	1,07	0,05028948
3100	1,07	1,07	0
3200	1,07	1,07	0
3300	1,061935	1,06	0,0040161
3400	1,061613	1,06	0,00734701
3500	1,06	1,06	0
3600	1,080645	1,06	0,11494739
3700	1,237742	1,06	0,30672148
3800	1,06	1,06	0

Par rapport à la version optimisée compilée via gcc – O2, prenons la valeur à 2500 on a encore un speedup:

Moy gcc02 opti / Moy iccO3

= 1,84 / 1,07

= 1,71

Nous avons donc un speed-up de 1,71 par rapport à la version optimisée

Et par rapport à la version initiale :

Moy init / Moy iccO3

= 3,24 / 1,07

= 3,028

Exécutons maintenant Maqao :

Global Metrics

Total Time (s)		14
Time in loops (%)		99.28
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.62
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.37
	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 (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 5	53-55	baseline:kernel.c	baseline	87.71	12.28	0	1	1.78	8
Loop 8	48-50	baseline:kernel.c	baseline	11.14	1.56	100	1	NA	NA
Loop 9	48-50	baseline:kernel.c	baseline	0.43	0.06	0	1	1	8

Nous voyons qu'il est encore possible d'optimiser. Essayons les autres.

2.2.2 GCC -O3 -march=native

En compilant avec GCC -O3 -march=native, on obtient les résultats suivants :

MOY 03nat	MED o3 nat	ECARTYPE
0,2803225806	0,28	0,001796053
0,3367741935	0,33	0,037717113
0,29	0,29	0
0,2806451613	0,28	0,00249731
0,28	0,28	0
0,28	0,28	0
0,28	0,28	0
0,2816129032	0,28	0,008980265
0,28	0,28	0
0,28	0,28	0
0,2761290323	0,28	0,004951376
0,32	0,32	0
0,2812903226	0,28	0,007184212
0,27	0,27	0
0,27	0,27	0
0,275483871	0,27	0,030532901
0,27	0,27	0
0,27	0,27	0
0,27	0,27	0

Par rapport à la version optimisée compilée via gcc – O2, prenons la valeur à 2500 on a encore un speedup:

Moy gcc02 opti / Moy gccO3

= 1,84 / 0,28

= 6,571

Nous avons donc un speed-up de 6,571 par rapport à la version optimisée

Et par rapport à la version initiale :

Moy init / Moy iccO3

= 3,24 / 0,28

= 11,57

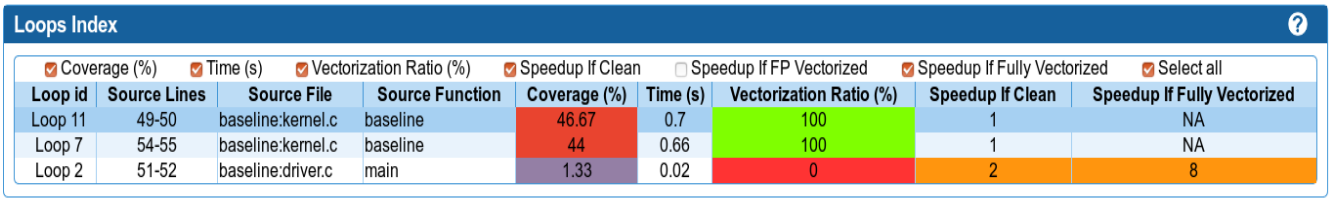
Exécutons maintenant Maqao :

Global Metrics

Total Time (s)		1.5
Time in loops (%)		92
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.36
Clean	Potential Speedup	1.01
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	1

Nous remarquons que le potentiel de speed-up est à 1,01 nous sommes donc presque au maximum.

Maqao nous indique aussi que l'option funroll-loops est manquant, nous allons l'ajouter par la suite.



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 Fully Vectorized
Loop 11	49-50	baseline:kernel.c	baseline	46.67	0.7	100	1	NA
Loop 7	54-55	baseline:kernel.c	baseline	44	0.66	100	1	NA
Loop 2	51-52	baseline:driver.c	main	1.33	0.02	0	2	8

Le ratio de vectorisation est à 100 % pour les deux premières boucles, c'est une bonne nouvelle car ces boucles prennes 46,67 % et 44 % de toutes les boucles.

2.2.3 gcc_-O3-march=native-funroll-loops-g-Wall

En compilant avec gcc_-Ofast-march=native-funroll-loops-g-Wall, on obtient les résultats suivants :

MOY ofast funloops	MED ofast funloops	ECARTYPE ofast funloops
0,266129032258064	0,25	0,0348976999581
0,26	0,26	0
0,26	0,26	0
0,25258064516129	0,25	0,010944630931948
0,25	0,25	0
0,25	0,25	0
0,25	0,25	0
0,25	0,25	0
0,240645161290323	0,24	0,002497310381147
0,25	0,25	0
0,25	0,25	0
0,250322580645161	0,25	0,001796053020268
0,250322580645161	0,25	0,001796053020268
0,25	0,25	0
0,24258064516129	0,24	0,004448027229746
0,25	0,25	0
0,24	0,24	0
0,25	0,25	0
0,24	0,24	0

Par rapport à la version optimisée compilée via gcc – O2, prenons la valeur à 2500 on a encore un speedup:

Moy gcc02 opti / Moy gccO3 native

= 1,84 / 0,25

= 7,36

Nous avons donc un speed-up de 6,571 par rapport à la version optimisée

Et par rapport à la version initiale :

Moy init / Moy gccO3 native

= 3,24 / 0,25

= 12,96

Exécutons maintenant Maqao :

Global Metrics

Total Time (s)		3.22
Time in loops (%)		93.16
Compilation Options		OK
Flow Complexity		1.01
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	1.01
	Nb Loops to get 80%	1

Loops Index

Name of the fonction that contains the loop.

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 11	49-50	baseline:kernel.c	baseline	46.58	1.5	100	1	NA	NA
Loop 7	54-55	baseline:kernel.c	baseline	45.34	1.46	100	1	NA	NA
Loop 10	48-50	baseline:kernel.c	baseline	1.24	0.04	0	1	1	8

Le potentiel de speed-up est à 1 ou 1,01. Nous pouvons encore optimiser une dernière fois.

2.2.4 gcc_-Ofast-march=native-funroll-loops-g-Wall

En compilant avec gcc_-Ofast-march=native-funroll-loops-g-Wall, on obtient les résultats suivants :

MOY ofast funloops	MED ofast funloops	ECARTYPE ofast funloops
0,240967741935484	0,24	0,003962186862224
0,213225806451613	0,21	0,017960530202678
0,240322580645161	0,24	0,001796053020268
0,200322580645161	0,2	0,001796053020268
0,240322580645161	0,24	0,001796053020268
0,200322580645161	0,2	0,001796053020268
0,241935483870968	0,24	0,009099214192705
0,190322580645161	0,19	0,001796053020268
0,250322580645161	0,25	0,001796053020268
0,192258064516129	0,19	0,009204954252776
0,240967741935484	0,24	0,003962186862224
0,211612903225806	0,21	0,007347005827115
0,240322580645161	0,24	0,001796053020268
0,190322580645161	0,19	0,001796053020268
0,24	0,24	0
0,190322580645161	0,19	0,001796053020268
0,24	0,24	0
0,191290322580645	0,19	0,005622535302317
0,24	0,24	0

Par rapport à la version optimisée compilée via gcc – O2, prenons la valeur à 2500 on a encore un speedup:

Moy gcc02 opti / Moy gccO3 ofast

= 1,84 / 0,2

= 9,2

Nous avons donc un speed-up de 6,571 par rapport à la version optimisée

Et par rapport à la version initiale :

Moy init / Moy gccO3 ofast

= 3,24 / 0,2

= 16,2

Exécutons maintenant Maqao :

Global Metrics

Total Time (s)		3.12
Time in loops (%)		92.95
Compilation Options		OK
Flow Complexity		1.00
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	1.00
	Nb Loops to get 80%	1

Nous somme au maximum d'optimisation, le flow-complexity est à 1, tous les potentiels sont à 1 et la vectorisation est à 100 %.

Loops Index

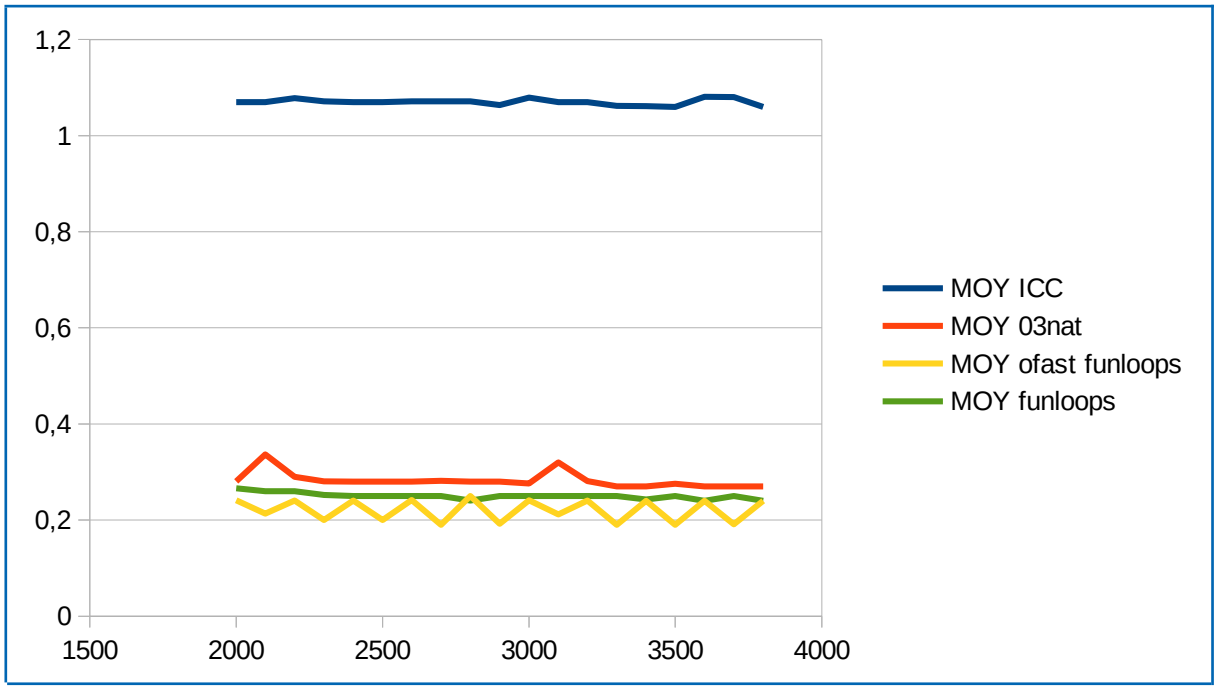
Name of the source file that contains the loop.

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

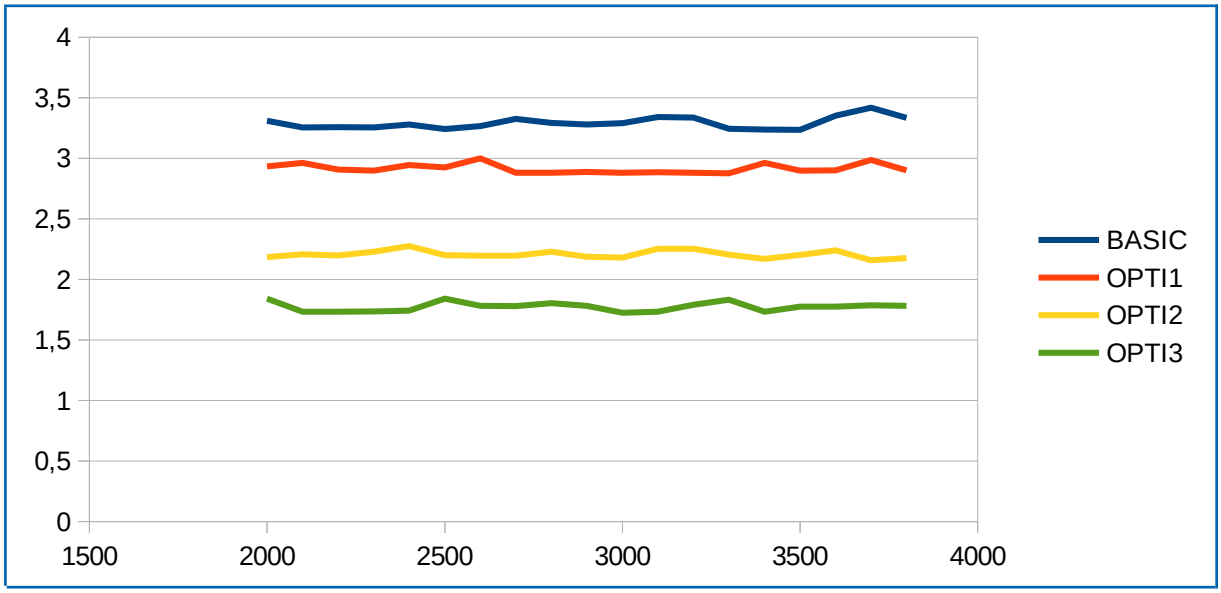
Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If Fully Vectorized
Loop 8	50-50	baseline:kernel.c	baseline	48.72	1.52	100	1	NA
Loop 6	54-55	baseline:kernel.c	baseline	44.23	1.38	100	1	NA

2.2.5 Bilan

Les options de compilations nous donnent cette courbe :

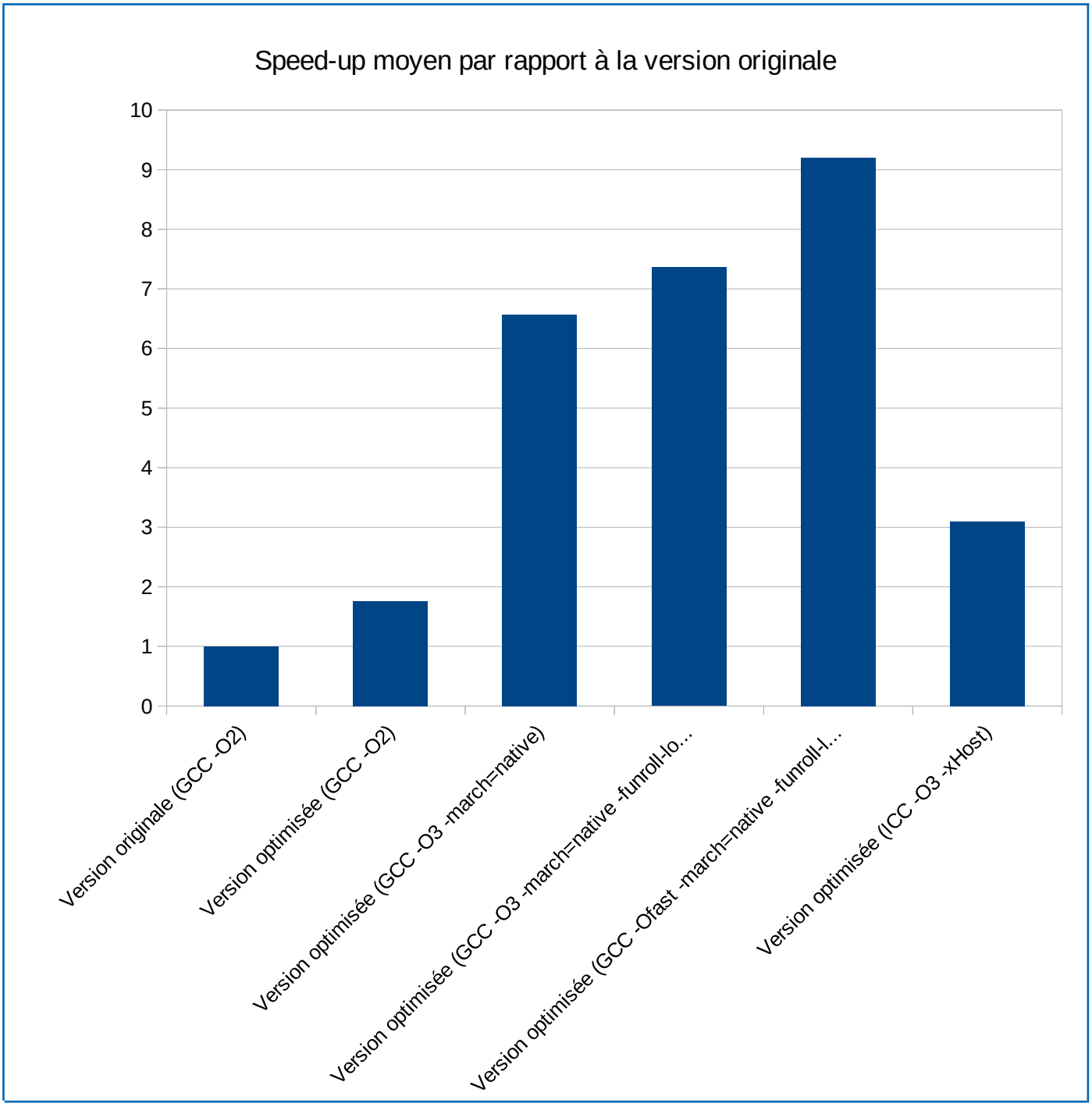


La dernière optimisation (en jaune) est donc la meilleure. Si nous comparons avec l'ancienne courbe de la version optimisée nous remarquons tout de suite la différence.



Ces résultats coïncident avec ceux de la phase 1, puisque déjà, ces deux degrés étaient les plus performants avec le code original.

On peut également comparer les speed-ups, pour accentuer le résultat précédent :



2.3 Parallélisation avec OpenMP

Enfin, on peut aussi paralléliser le noyau, à l'aide de la librairie OpenMP.
Cette bibliothèque permet de paralléliser des portions de codes, à l'aide de pragmas.

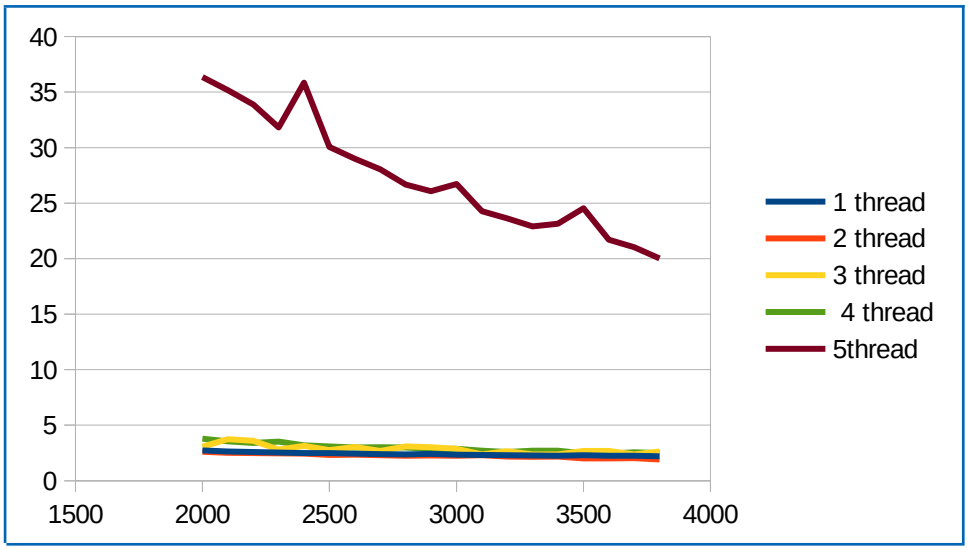
Ici, on ne possède que deux boucles for, on peut donc paralléliser le noyau de la manière suivante :

```
/* 4ème version : parallélisée avec OpenMP */
#include <omp.h>  // On inclut la librairie OpenMP
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        #pragma omp parallel for
        for (i = 0; i < n/2; i++) {
                b[i] = (a[i] > x ? a[i] : x);
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }

        #pragma omp parallel for
        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }
}
```

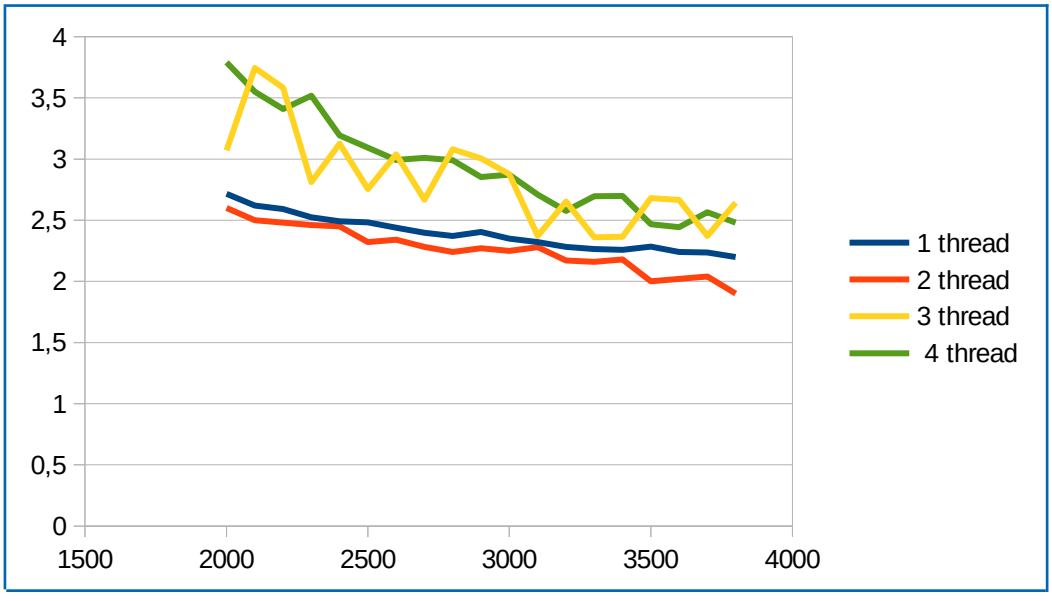
2.3.1 Choix du nombre de threads

Concernant le compilateur, nous avons ici repris notre compilateur de base (gcc -O2).
Une des particularités de la parallélisation réside dans le nombre de threads à utiliser. Pour ce faire, nous avons utilisé un script (*omp.sh*), qui a exécuté le programme pour 1 à 5 threads.

On a ainsi obtenu la courbe suivante :



Dans un premier temps cette courbe ne nous informe pas grand-chose à part que 5 threads n'est pas du tout intéressant. Nous pouvons donc supprimer la courbe représentant les 5 threads et nous concentrer sur les 4 autres :



On remarque que le temps d'exécution du programme est optimal pour 2 threads. En effet, au-delà de 2 threads, le programme s'exécute plus lentement.

Partitionner une tâche entre un nombre élevé de threads coûte relativement cher (la synchronisation entre les threads notamment).
De plus, plus il y a de threads, moins les threads ont de travail à effectuer, et ont un coût non-négligeable pour le système.

Pour pallier à cela, il est donc important de choisir judicieusement le nombre de threads, par exemple via une courbe comme ci-dessus.
Attention toutefois, cette limite est différente sur chaque machine puisqu'elle dépend du processeur considéré.

2.3.2 Mesures

Pour n = 2500, et 2 threads on a :

Moyenne (cycles/itérations)	2,32
Minimum (cycles/itérations)	2,30
Maximum (cycles/itérations)	2,33
Médiane (cycles/itérations)	2,325

On remarque dans un premier temps sont légèrement moins régulières que celles mesurées précédemment. Ceci est certainement dû à l'utilisation de la librairie OpenMP.

Si l'on compare à la version optimisée (sans parallélisation), compilée avec gcc -O2 (compilateur de base), on a :

$$G_{moyen}(V_{OMP}(gcc-O2)/V_{OPT3}(gcc-O2)) \approx \frac{1,84}{2,32} \approx 0,836$$

La version parallélisée est donc 0,836 fois moins rapide que la version optimisée non-parallélisée. Le gain de performances est réduit. Ce n'est donc pas rentable sur ce cache avec cette machine.

3 Travaux sur le cache L2 – Clément LEFEVRE

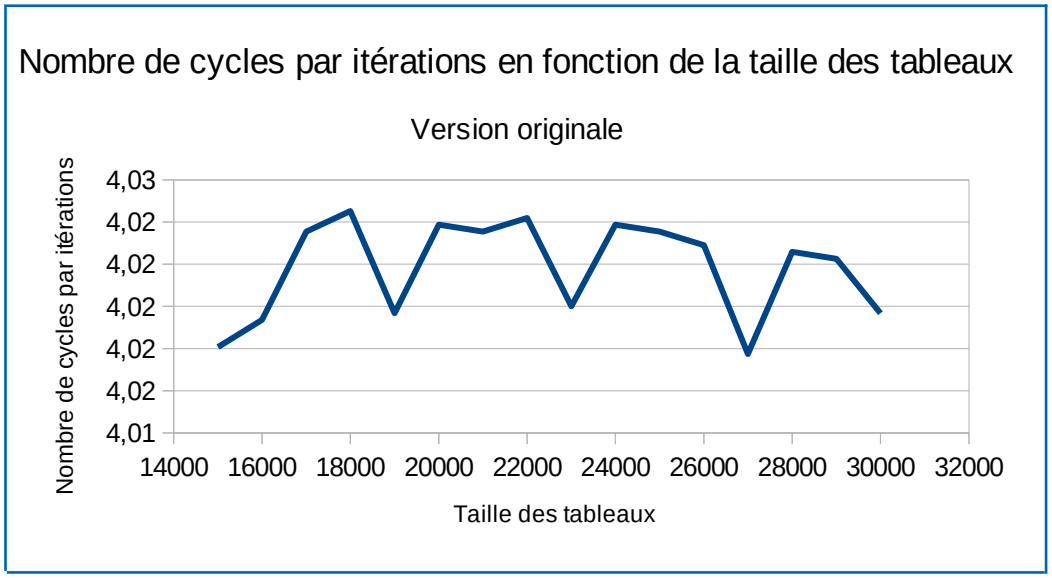
3.1 Optimisations du code-source

3.1.1 Version originale

```
/* Original */
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;
    }
}
```

En exécutant la version originale du programme, compilée avec GCC -O2, notre compilateur de référence, avec n = 25000, on obtient les résultats suivants :

Moyenne (cycles/itérations)	4,02
Minimum (cycles/itérations)	4,01
Maximum (cycles/itérations)	4,05
Médiane (cycles/itérations)	4,02
Écart-type (cycles/itérations)	0,01



En analysant les résultats fournis par MAQAO, notamment les métriques principales, on a :

Global Metrics

Total Time (s)		16.4
Time in loops (%)		100
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	6.99
	Nb Loops to get 80%	2

Comme GCC -O2 n'inclue pas le déroulage de boucle, MAQAO réclame le flag correspondant.

On remarque également que la complexité du flux est trop élevée (2.00).

Enfin, avec un code vectorisé, MAQAO nous indique que l'on pourrait obtenir un speed-up de quasiment 7.

Au niveau des boucles, on remarque que la complexité des deux boucles est importante, et que la quasi-totalité du temps d'exécution du programme se passe dans ces deux boucles (52.07% + 47.93%).

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 6	151-157	baseline:kernel.c	baseline	52.07	8.54	0	1.25	1.6	6.46
Loop 5	160-162	baseline:kernel.c	baseline	47.93	7.86	0	1	1	7.43

Une des premières choses marquante, est que ces deux boucles ne sont absolument pas vectorisées (ratio de vectorisation de 0%). Le speed-up potentiel obtenu après vectorisation est donc élevé (x7 environ).

Si l'on s'intéresse plus précisément au code-source, l'objet de nos optimisations pour cette phase n°2, pour la première boucle, on a :

```
151:     for (i = 0; i < n; i++) {
152:          if ((i < n/2) && (a[i] > x))
153:               b[i] = a[i];
154:          else if (i < n/2)
155:               b[i] = x;
156:          else
157:               b[i] = a[i] + x;
158:     }
```

Coverage 52.07 %
Function baseline
Source file and lines kernel.c:151-157
Module baseline
The loop is defined in /home/clement/Documents/ISTY/IATIC4/ProjetAOA/kernel.c:151-157.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

Ex: if (x<0) x=0 => x = (x<0 ? 0 : x) (or MAX(0,x) after defining the corresponding macro)

MAQAO a parfaitement reconnu la structure de cette fonction (<if then [else] end>) et nous indique que cette structure et le fait qu'il existe plusieurs chemins d'exécutions (du fait de cette structure) constitue le principal goulot d'étranglement (*bottleneck*).
Pour pallier à cela, MAQAO nous propose les alternatives suivantes :

- Sortir la condition de la boucle
- Utiliser des expression logiques (comme dans l'exemple).

Les recommandations sont les mêmes pour la seconde boucle, puisque elle-aussi contient une expression conditionnelle en son intérieur.

```
160:     for (i = 0; i < n; i++) {
161:          if (b[i] < 0.0)
162:               b[i] = 0.0;
163:     }
```

En terme de travail annexe, MAQAO nous invite à changer l'option de compilation en -O3, afin d'autoriser la vectorisation, néanmoins, ce n'est pas le but de cette phase donc nous n'allons pas le faire pour le moment.

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)

Nous ne sommes également pas concernés par la deuxième proposition d'amélioration, puisque nous ne manipulons ni structure de données ni tableaux à deux dimensions.

3.1.2 1ère optimisation (OPT1), « If Hoisting »

Dans un premier temps, on peut remarquer qu'une des conditions de la première boucle for contient une condition sur la valeur de *n*, qui ne dépend donc pas de la boucle.

```
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;
```

```
 }
```

On peut donc sortir cette condition de la boucle, de la manière suivante :

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

    for (i = n/2; i < n; i++) {
         b[i] = a[i] + x;
    }

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

La première boucle for, de taille n a donc été décomposée en deux boucles de taille n/2. Toutefois, il reste des expressions conditionnelles, mais celles-ci ne sont pas sortables de la boucles, car dépendant directement de i. En revanche, on pourrait tester de fusionner la dernière boucle avec les autres, ce sera l'objet de la prochaine optimisation.

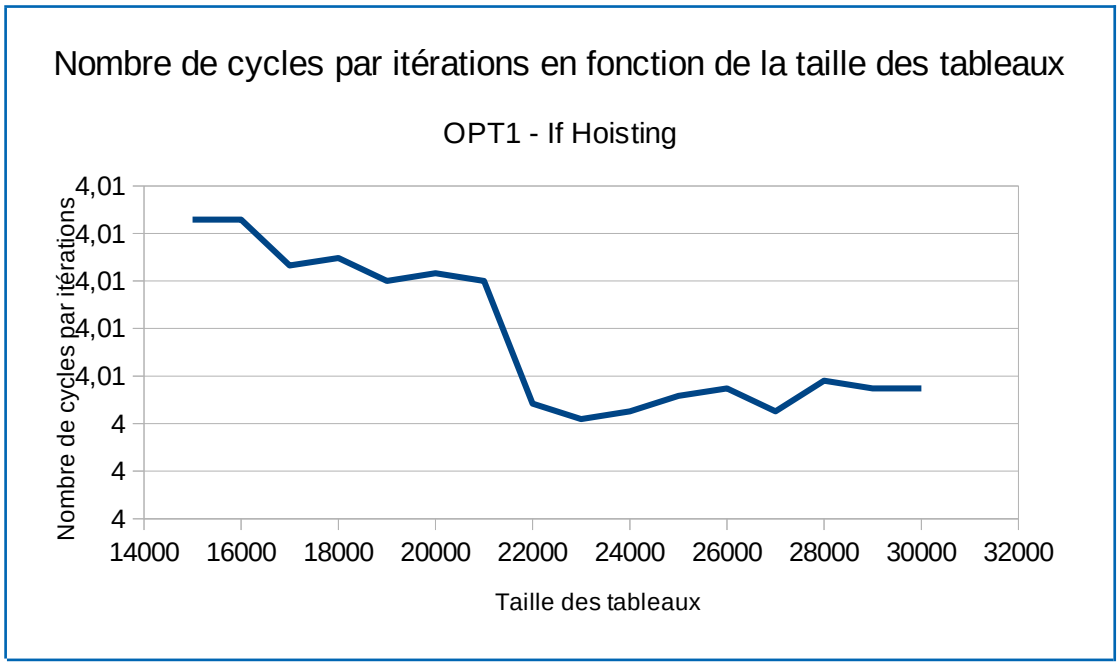
En compilant cette première version optimisée avec GCC -O2, notre compilateur de référence, et en l'exécutant avec n = 25000, on obtient les résultats suivants :

Moyenne (cycles/itérations)	4,01
Minimum (cycles/itérations)	4,00
Maximum (cycles/itérations)	4,02
Médiane (cycles/itérations)	4,00
Écart-type (cycles/itérations)	0,01

On remarque que ces valeurs sont sensiblement identiques à celles mesurées pour la version originale (en réalité elles sont très légèrement meilleures).

$$G_{moyen}(V_{OPT1}/V_{Originale}) \approx \frac{4.02}{4.01} \approx 1.00$$

$$G_{médian}(V_{OPT1}/V_{Originale}) \approx \frac{4.02}{4.00} \approx 1.01$$



Lorsque l'on analyse le bilan fourni par MAQAO, on observe les résultats suivants :

La complexité du flux est améliorée (1,49 contre 2,00 dans la version originale).

Le speed-up avec un code vectorisé serait de 7,30.

Global Metrics

Total Time (s)		30.26
Time in loops (%)		99.8
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.49
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.15
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.30
	Nb Loops to get 80%	3

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	16-18	baseline:kernel.c	baseline	49.11	14.86	0	1	1	7.43
Loop 6	12-13	baseline:kernel.c	baseline	25.64	7.76	0	1	2	8
Loop 7	5-9	baseline:kernel.c	baseline	24.98	7.56	0	1	1	8
Loop 4	13-14	baseline:driver.c	main	0.07	0.02	25	1.5	2.13	7.11

Comme nous l'avons vu précédemment, cette version optimisée possède trois boucles :deux de taille n/2 et une de taille n, ce qui est visible ci-dessus (la boucle n°5 couvre environ 50 % du temps d'exécution et les boucles n°6 et 7 environ 25 % chacune).

Toutes ces boucles pourraient être vectorisées, ce qui offrirait un speed-up important au programme.

Concernant les recommandations, pour la boucle n°5 (qui est en fait la dernière boucle de la fonction), MAQAO nous propose les mêmes axes d'améliorations que précédemment.

Coverage 49.11 %
Function baseline
Source file and lines kernel.c:16-18
Module baseline
The loop is defined in /home/clement/Documents/ISTY/IATIC4/ProjetAOA/kernel.c:16-18.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

Ex: if (x<0) x=0 => x = (x<0 ? 0 : x) (or MAX(0,x) after defining the corresponding macro)

Or, pour cette boucle, on ne peut sortir la condition (i.e. effectuer un hoisting). On va donc (pour les futures optimisations) appliquer la seconde proposition : modifier cette condition en une affectation conditionnelle (à la manière de l'exemple ci-dessus, qui correspond d'ailleurs exactement à notre cas).

Pour les deux autres boucles, MAQAO nous recommande simplement d'effectuer du déroulage de boucle, néanmoins, pour la suite, nous allons essayer de remplacer les expressions conditionnelles par des affectations conditionnelles (comme ci-dessus).

3.1.3 2ème optimisation (OPT2) – « If Hoisting & Loop fusion »

Pour cette deuxième version, je n'ai pas souhaité implémenter de suite les recommandations de MAQAO, mais plutôt effectuer une optimisation qui me semblait évidente au vu du précédent code : la fusion de boucles (cumulée au hoisting).

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

        if(b[i] < 0.0)
            b[i] = 0.0;
    }

    for (i = n/2; i < n; i++){
        b[i] = a[i] + x;

        if(b[i] < 0.0)
            b[i] = 0.0;
    }
}
```

On a donc fusionné la dernière boucle (de taille n) avec les deux premières (de taille n/2). La complexité est donc désormais de n (contre 2n précédemment).

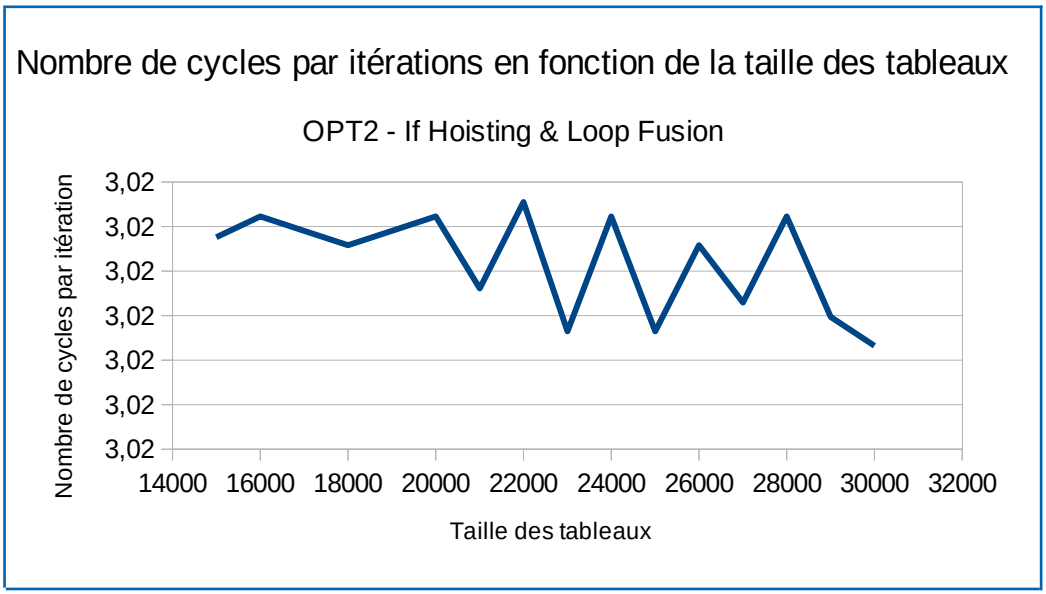
Toutefois, il faut tout de même veiller à conserver l'ordre des instructions, pour conserver le sens du programme (la condition ajoutée sur b doit être effectuée après son affectation).

Pour n = 25000, on a :

Moyenne (cycles/itérations)	3,02
Minimum (cycles/itérations)	3,02
Maximum (cycles/itérations)	3,03
Médiane (cycles/itérations)	3,02
Écart-type (cycles/itérations)	0,00

On constate une réelle amélioration du temps d'exécution par rapport aux deux précédentes versions de la fonction.

$$G_{median}(V_{OPT\,2}/V_{Originale}) \approx G_{moyen}(V_{OPT\,2}/V_{Originale}) \approx \frac{4.02}{3.02} \approx 1.33$$



Si l'on analyse le bilan fourni par MAQAO, on remarque que, bien que l'exécution soit plus rapide (22.84s contre 30.26s précédemment), la complexité du flux augmente elle drastiquement (2.97 ici).

Ceci peut s'expliquer par le fait que nos boucles soient relativement chargées d'expressions conditionnelles (if).

Global Metrics

Total Time (s)		22.84
Time in loops (%)		99.83
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.97
Array Access Efficiency (%)		70.89
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.18
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	6.50
	Nb Loops to get 80%	2

Concernant les boucles, on observe 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 (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 6	26-33	baseline:kernel.c	baseline	65.59	14.98	7.74	1.08	1	5.88
Loop 5	36-40	baseline:kernel.c	baseline	34.24	7.82	20	1	1.78	8

La première boucle (n°6 ici), possédant la structure <if> <else> <if> est celle qui occupe la majeure partie du temps d'exécution (65.69 %).

Du fait de cette structure relativement complexe, MAQAO nous informe qu'il serait donc préférable de sortir les expressions conditionnelles de la boucle, ou de les transformer en affectations conditionnelles (ce que nous allons faire pour la prochaine optimisation).

3.1.4 3ème optimisation (OPT3) – Affectations conditionnelles

Comme conseillé par MAQAO depuis la version originale, nous allons ici mettre en place des affectations conditionnelles en remplacement des expressions conditionnelles.

On peut notamment remplacer l'expression conditionnelle :

```
if (a[i] > x)
    b[i] = a[i];
else
    b[i] = x;
```

Par l'affectation conditionnelle équivalente :

```
b[i] = (a[i] > x ? a[i] : x)
```

L'expression conditionnelle est évaluée, et si elle est vérifiée, a[i] sera affecté à b, sinon ce sera x.

On procède de la même manière pour l'ensemble de la fonction, et, en prenant toujours garde à conserver l'ordre des instructions et leurs dépendances, à partir de la dernière version optimisée, on arrive donc à la version ci-après :

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

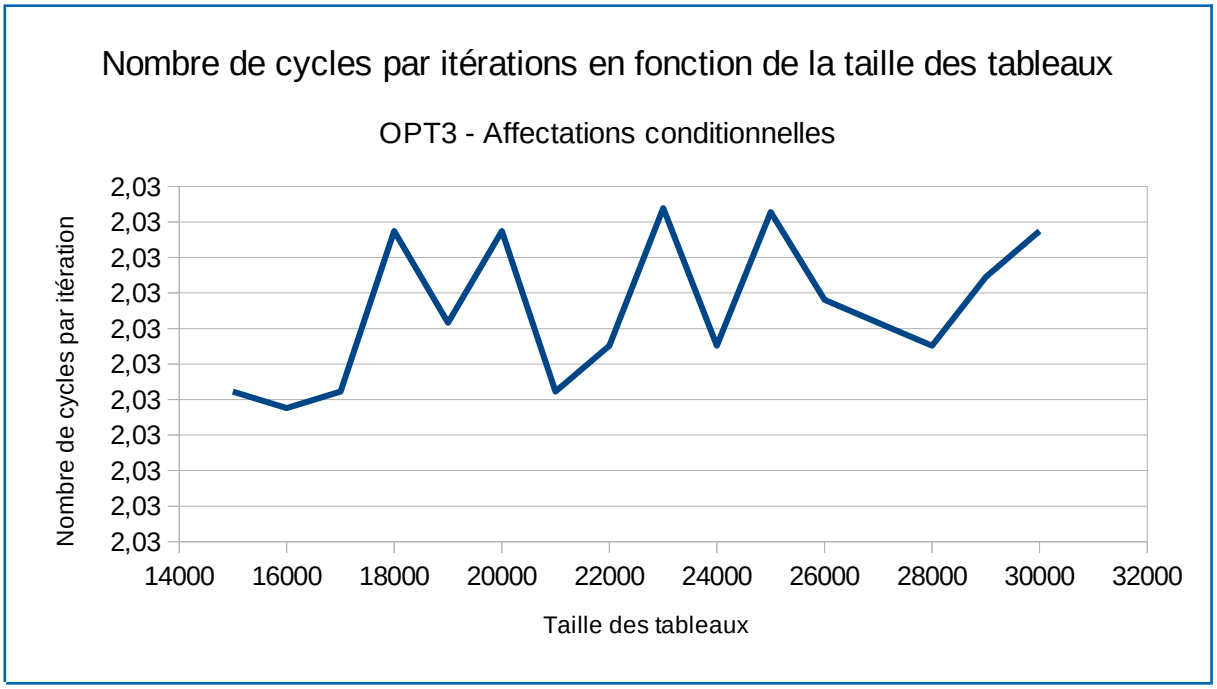
```
        b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
    }

    for (i = n/2; i < n; i++){
        b[i] = a[i] + x;
        b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
    }
}
```

Pour n = 25000, on obtient les valeurs suivantes :

Moyenne (cycles/itérations)	2,03
Minimum (cycles/itérations)	2,03
Maximum (cycles/itérations)	2,06
Médiane (cycles/itérations)	2,03
Écart-type (cycles/itérations)	0,01

$$G_{median}(V_{OPT3}/V_{Originale}) \approx G_{moyen}(V_{OPT3}/V_{Originale}) \approx \frac{4.02}{2.03} \approx 1.98$$

Cette version est donc quasiment deux fois plus rapide que la version originale.



Lorsque l'on analyse les métriques globales de MAQAO, on remarque que le temps total d'exécution est beaucoup plus court (seulement 15.38s, contre 22.84s pour la version précédente).

De plus, la complexité du flux est optimale (1,00).

MAQAO nous indique également que l'on pourrait atteindre encore un speed-up de 7.65 avec un code totalement vectorisé.

Global Metrics

Total Time (s)		15.38
Time in loops (%)		99.35
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
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	7.65
	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 (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 6	48-50	baseline:kernel.c	baseline	51.5	7.92	20	1	1	8
Loop 5	53-55	baseline:kernel.c	baseline	47.85	7.36	20	1	1.78	8

Grâce à l'optimisation effectuée, les boucles sont équilibrées (elles couvrent chacune environ 50 % du temps d'exécution). Elles sont également vectorisées à hauteur de 20 %, ce qui pourrait encore être amélioré.

Hormis la vectorisation, que nous allons autoriser via les différentes options de compilations dans la partie suivante, MAQAO ne nous fait aucune autre recommandation sur les boucles.

Vectorization

Your loop is probably not vectorized. Only 20% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 2.00 to 0.25 cycles (8.00x speedup).

Details

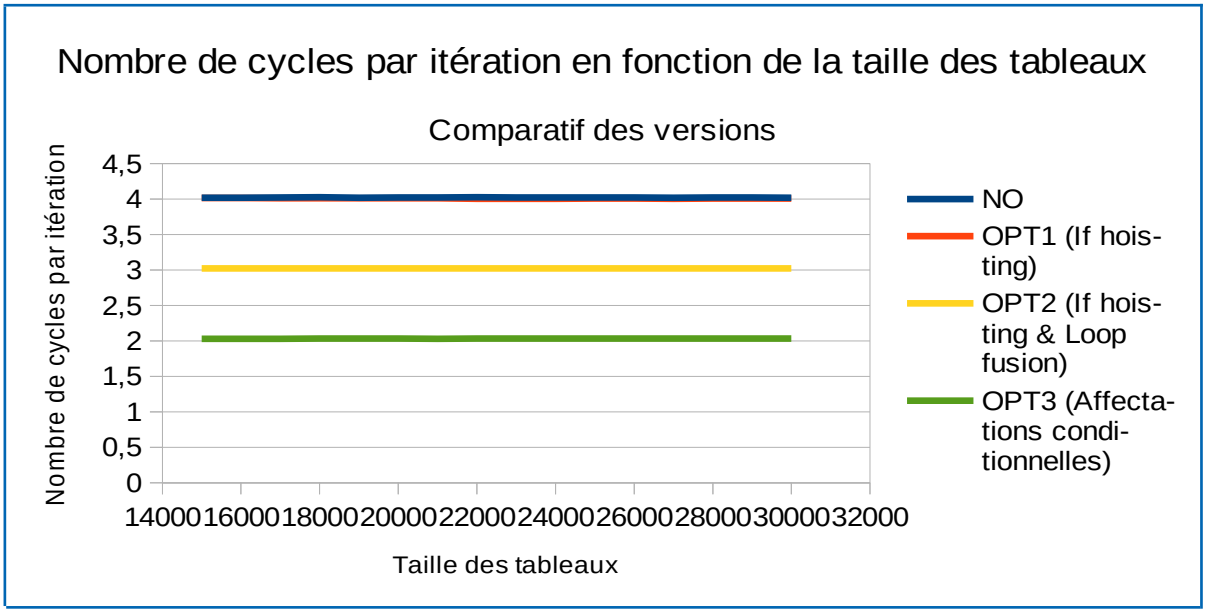
Store and arithmetical SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

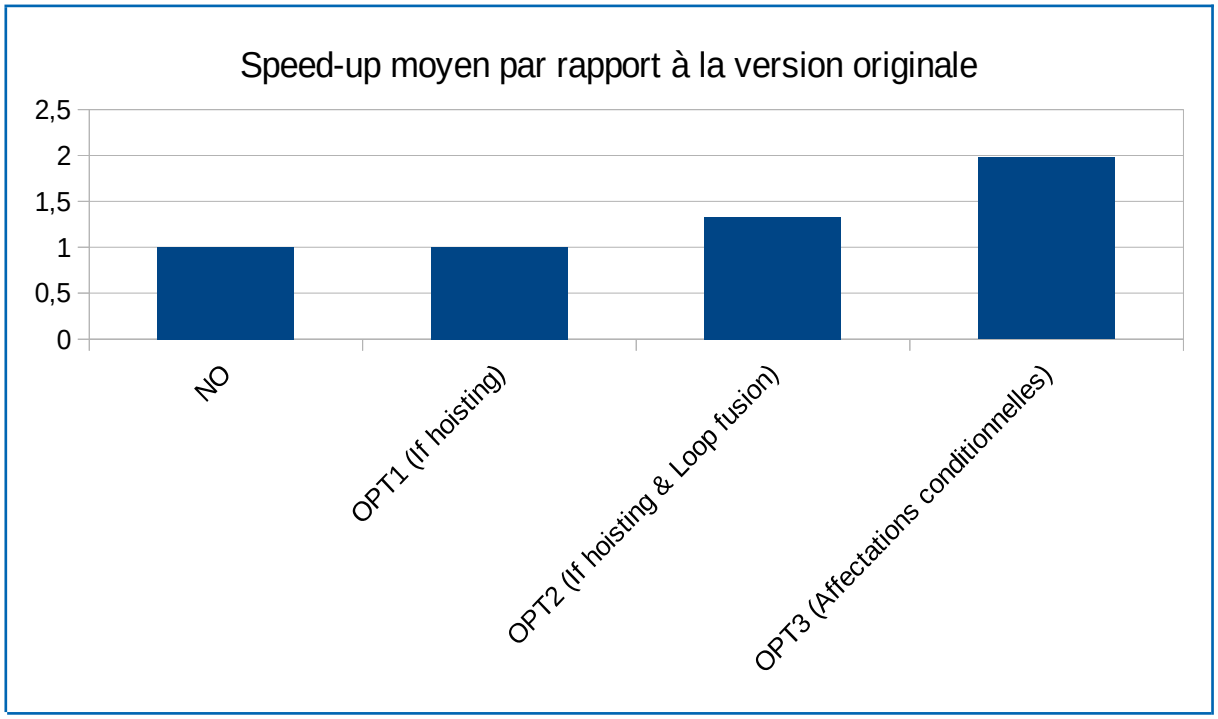
- Try another compiler or update/tune your current one:
  - recompile with ftree-vectorize (included in O3) to enable loop vectorization and with fassociative-math (included in Ofast or ffast-math) to extend vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make 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)

3.1.5 Bilan

Si l'on compare toutes les versions jusqu'à présent, on obtient la courbe suivante :



L'optimisation 3 est donc naturellement la plus performante. On peut également regarder les speed-ups de chaque versions par rapport à la version originale :



3.2 Utilisation d'optimisations plus agressives

On va maintenant modifier l'optimisation de base (GCC -O2) avec notre code source optimisé, et tester d'autres degrés d'optimisations.

Au sein de la première phase du projet, nous avions pu remarquer que l'optimisation GCC -O3 -march=native, avec le code-source de base, produisait le meilleur résultat en termes cycles par itérations.
On va considérer ici uniquement les 4 optimisations les plus efficaces lors de la phase 1 sur cette machine, à savoir, du moins efficace au plus efficace :

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

3.2.1 ICC -O3 -xHost

En compilant avec icc -O3 -xHost, pour n = 25000, on obtient les résultats suivants :

Moyenne (cycles/itérations)	1,29
Minimum (cycles/itérations)	1,29
Maximum (cycles/itérations)	1,32
Médiane (cycles/itérations)	1,29
Écart-type (cycles/itérations)	0,01

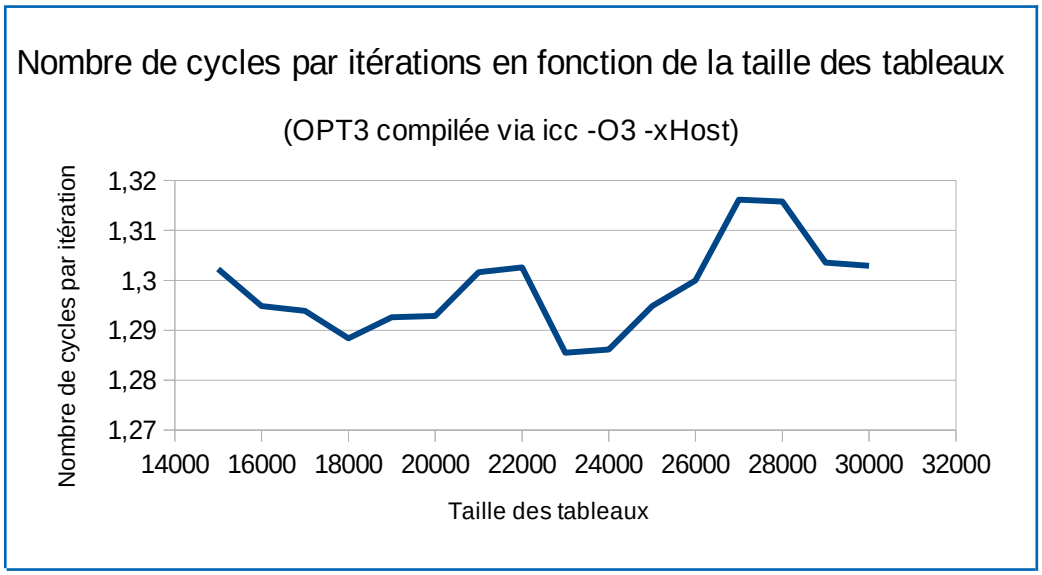
Par rapport à la version optimisée compilée via gcc -O2, on a :

$$G_{moyen}(V_{OPT3}(icc-O3-xHost)/V_{OPT3}(gcc-O2)) \approx \frac{2.03}{1.29} \approx 1.57$$

Et par rapport à la version initiale :

$$G_{moyen}(V_{OPT3}(icc-O3-xHost)/V_{Originale}(gcc-O2)) \approx \frac{4.02}{1.29} \approx 3.12$$

Cette version est donc environ trois fois plus rapide que la fonction originale.



En analysant les résultats avec MAQAO, on remarque que le compilateur vectorise à 100 % la première boucle (ici n°5), mais pas la seconde (n°7).

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	53-55	baseline:kernel.c	baseline	77.05	102.2	0	1	1.71	8
Loop 7	48-50	baseline:kernel.c	baseline	22.81	30.26	100	1	1	1
Loop 9	48-50	baseline:kernel.c	baseline	0.05	0.06	0	1	1	8
Loop 2	51-52	baseline:driver.c	main	0.02	0.02	0	3	1	8

Le speed-up n'est donc pas encore optimal, on va donc tester d'autres optimisations, notamment avec GCC.

3.2.2 GCC -O3 -march=native

En compilant avec gcc -O3 -march=native, pour n = 25000, on obtient les résultats suivants :

Moyenne (cycles/itérations)	0,63
Minimum (cycles/itérations)	0,62
Maximum (cycles/itérations)	0,65
Médiane (cycles/itérations)	0,63
Écart-type (cycles/itérations)	0,01

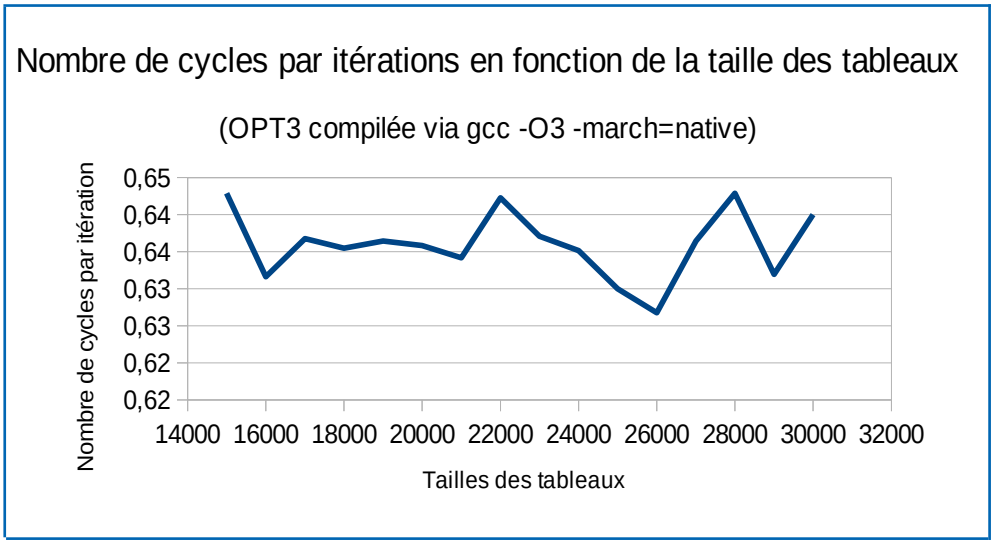
Par rapport à la version optimisée et compilée via gcc -O2, on a :

$$G_{moyen}(V_{OPT3}(gcc-O3-march=native)/V_{OPT3}(gcc-O2)) \approx \frac{2.03}{0.63} \approx 3.22$$

Et par rapport à la version originale :

$$G_{moyen}(V_{OPT3}(gcc-O3-march=native)/V_{Originale}(gcc-O2)) \approx \frac{4.02}{0.63} \approx 6.38$$

Cette version est donc plus de 6 fois plus rapide que la version originale.



En analysant les résultats fournis par MAQAO, on remarque que le code est beaucoup plus vectorisé que dans les versions précédentes, puisque le speed-up potentiel si le code est totalement vectorisé est d'uniquement 1,12.

Global Metrics

Total Time (s)		63.1
Time in loops (%)		99.56
Compilation Options		binary: -funroll-loops 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	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.12
	Nb Loops to get 80%	2

MAQAO préconise cependant l'utilisation du flag -funroll-loops, permettant au compilateur d'effectuer du déroulage de boucle.

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	54-55	baseline:kernel.c	baseline	50.43	31.82	100	1	1	1.12
Loop 11	49-50	baseline:kernel.c	baseline	49	30.92	100	1	1	1.11
Loop 10	48-50	baseline:kernel.c	baseline	0.1	0.06	0	1.5	1	7.56
Loop 2	51-52	baseline:driver.c	main	0.03	0.02	0	3	1	8

Les boucles sont donc vectorisées totalement (ration de vectorisation de 100%).

On peut vérifier cela en regardant le code assembleur généré, par exemple pour la première fonction (n°7), on a :

```
0xeb0 VADDPS (%R10,%RAX,1),%YMM4,%YMM2
0xeb6 ADD $0x1,%ECX
0xeb9 VCMPPS $0x1,%YMM3,%YMM2,%YMM1
0xebe VANDNPS %YMM2,%YMM1,%YMM1
0xec2 VMOVUPS %XMM1,(%R8,%RAX,1)
0xec8 VEXTRACTF128 $0x1,%YMM1,0x10(%R8,%RAX,1)
0xed0 ADD $0x20,%RAX
0xed4 CMP %ECX,%R11D
0xed7 JA eb0 <baseline+0x330>
```

Ce code utilise de nombreuses instructions vectorielles (VADDPS, VCMPPS, etc.)

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

Comme nous l'a recommandé MAQAO, on va simplement ajouter le flag -funroll-loops.
Pour n = 25000, on a :

Moyenne (cycles/itérations)	0,60
Minimum (cycles/itérations)	0,59
Maximum (cycles/itérations)	0,62
Médiane (cycles/itérations)	0,60
Écart-type (cycles/itérations)	0,01

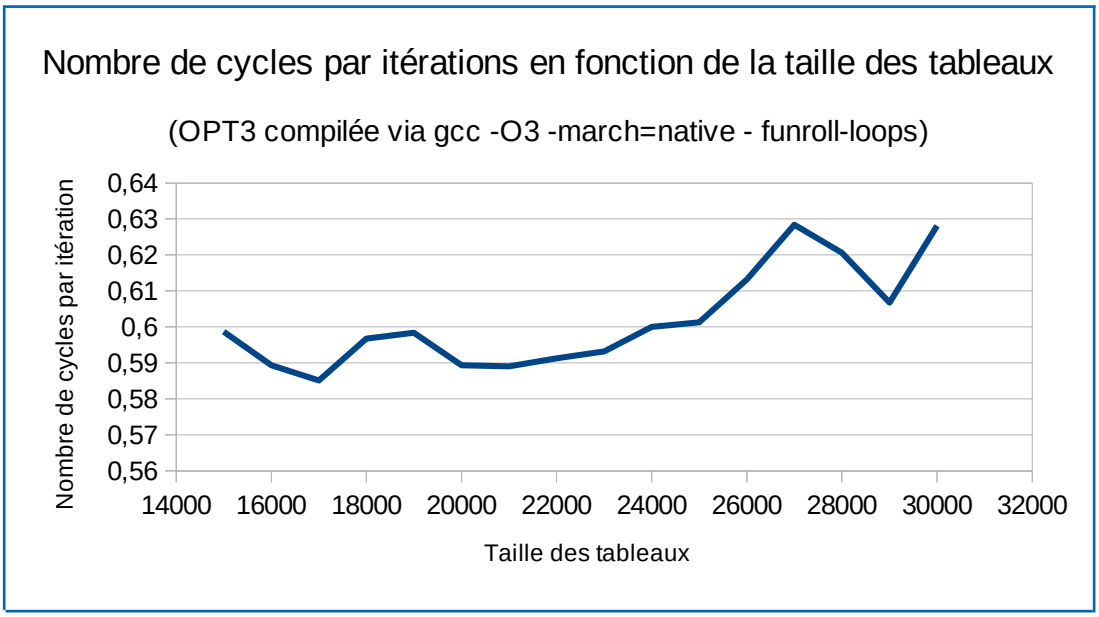
Par rapport à la version optimisée, compilée via gcc -O2, on a :

$$G_{moyen}(V_{OPT3}(gcc-O3-march=native-funroll-loops)/V_{OPT3}(gcc-O2))\approx\frac{2.03}{0.60}\approx 3.38$$

Et par rapport à la version originale :

$$G_{moyen}(V_{OPT3}(gcc-O3-march=native-funroll-loops)/V_{Originale}(gcc-O2))\approx\frac{4.02}{0.60}\approx 6.7$$

Le speed-up est donc encore légèrement amélioré (la version optimisée 6.7 fois plus rapide que la version originale).



Cette fois, les métriques de MAQAO sont toutes au vert, on se rend donc compte que nous arrivons bientôt au termes des optimisations réalisables.

Global Metrics

Total Time (s)		59.58
Time in loops (%)		99.25
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.01
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.06
	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 (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 7	54-55	baseline:kernel.c	baseline	50.59	30.14	100	1	1	1.06
Loop 11	49-50	baseline:kernel.c	baseline	48.47	28.88	100	1.02	1	1.05
Loop 10	48-50	baseline:kernel.c	baseline	0.13	0.08	0	1.5	1	7.56
Loop 4	13-14	baseline:driver.c	main	0.03	0.02	25	1.5	1.57	6.75
Loop 2	51-52	baseline:driver.c	main	0.03	0.02	0	3.22	1	8

On notera toutefois l'impact du flag -funroll-loops sur le code assembleur généré :

```
0x19e6 VADDPS (%R8,%RAX,1),%YMM13,%YMM12
0x19ec ADD $0x8,%R9D
0x19f0 VCMPPS $0x1,%YMM14,%YMM12,%YMM1
0x19f6 VANDNPS %YMM12,%YMM1,%YMM15
0x19fb VMOVUPS %XMM15,(%R13,%RAX,1)
0x1a02 VEXTRACTF128 $0x1,%YMM15,0x10(%R13,%RAX,1)
0x1a0a VADDPS 0x20(%R8,%RAX,1),%YMM13,%YMM2
0x1a11 VCMPPS $0x1,%YMM14,%YMM2,%YMM3
0x1a17 VANDNPS %YMM2,%YMM3,%YMM4
0x1a1b VMOVUPS %XMM4,0x20(%R13,%RAX,1)
0x1a22 VEXTRACTF128 $0x1,%YMM4,0x30(%R13,%RAX,1)
0x1a2a VADDPS 0x40(%R8,%RAX,1),%YMM13,%YMM5
0x1a31 VCMPPS $0x1,%YMM14,%YMM5,%YMM6
0x1a37 VANDNPS %YMM5,%YMM6,%YMM7
0x1a3b VMOVUPS %XMM7,0x40(%R13,%RAX,1)
0x1a42 VEXTRACTF128 $0x1,%YMM7,0x50(%R13,%RAX,1)
// ETC. (8 fois en tout)
0x1b11 ADD $0x100,%RAX
0x1b17 CMP %R9D,%R14D
0x1b1a JA 19e6 <baseline+0x7a6>
```

On remarquera que le code a été déroulé par 8, et vectorisé par 8, ce qui signifie qu'en une itération sont traitées 64 tours de boucles.

3.2.4 GCC -Ofast -march=native -funroll-loops

Il peut être judicieux de tester les mêmes flags avec le degré d'optimisation supérieur à -O3 : -Ofast.
Pour n = 25 000, on a :

Moyenne (cycles/itérations)	0,60
Minimum (cycles/itérations)	0,59
Maximum (cycles/itérations)	0,61
Médiane (cycles/itérations)	0,60
Écart-type (cycles/itérations)	0,01

Les résultats sont donc sensiblement les mêmes que pour l'optimisation précédente.

Pour prendre connaissance des différences, on peut regarder du côté de MAQAO :

Global Metrics

Total Time (s)		59.8
Time in loops (%)		99.09
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.05
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

Ce degré d'optimisation semble être optimal, puisque les métriques sont encore meilleures que pour gcc -O3 -funroll-loops -march=native.

Loops Index

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

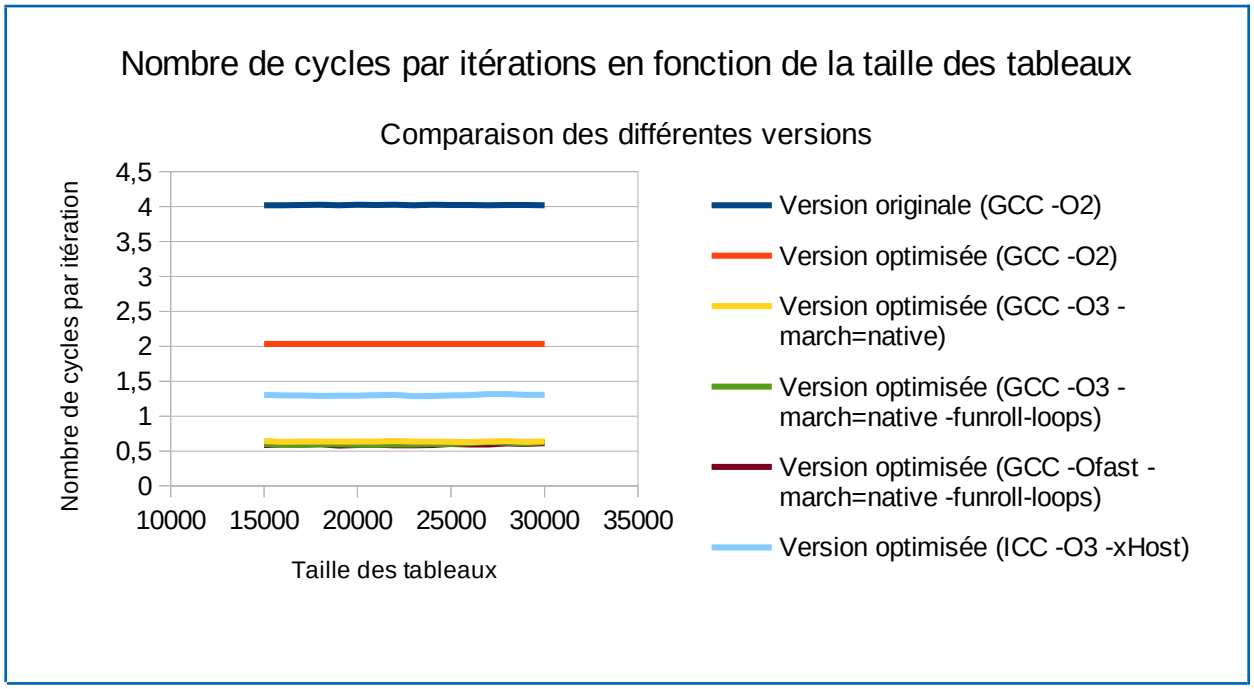
Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 8	50-50	baseline:kernel.c	baseline	49.46	29.58	100	1	1	1
Loop 6	54-55	baseline:kernel.c	baseline	49.43	29.56	100	1	1	1
Loop 2	51-52	baseline:driver.c	main	0.2	0.12	0	3.22	1	8

Le jeu d'instructions utilisé diffère légèrement du précédent, puisque l'instruction VCMPPS a été remplacée par VMAXPS.

```
0x1a39 VADDPS (%R8,%RAX,1),%YMM10,%YMM3    [1]
0x1a3f ADD $0x8,%R10D
0x1a43 VMAXPS %YMM2,%YMM3,%YMM4
0x1a47 VMOVUPS %XMM4,(%RCX,%RAX,1)    [2]
0x1a4c VEXTRACTF128 $0x1,%YMM4,0x10(%RCX,%RAX,1)    [2]
0x1a54 VADDPS 0x20(%R8,%RAX,1),%YMM10,%YMM5    [1]
0x1a5b VMAXPS %YMM2,%YMM5,%YMM6
0x1a5f VMOVUPS %XMM6,0x20(%RCX,%RAX,1)    [2]
0x1a65 VEXTRACTF128 $0x1,%YMM6,0x30(%RCX,%RAX,1)    [2]
0x1a6d VADDPS 0x40(%R8,%RAX,1),%YMM10,%YMM7    [1]
0x1a74 VMAXPS %YMM2,%YMM7,%YMM8
0x1a78 VMOVUPS %XMM8,0x40(%RCX,%RAX,1)    [2]
0x1a7e VEXTRACTF128 $0x1,%YMM8,0x50(%RCX,%RAX,1)    [2]
0x1a86 VADDPS 0x60(%R8,%RAX,1),%YMM10,%YMM9    [1]
0x1a8d VMAXPS %YMM2,%YMM9,%YMM11
```

3.2.5 Bilan

Si l'on compare tous les degrés d'optimisations, on obtient la courbe suivante :

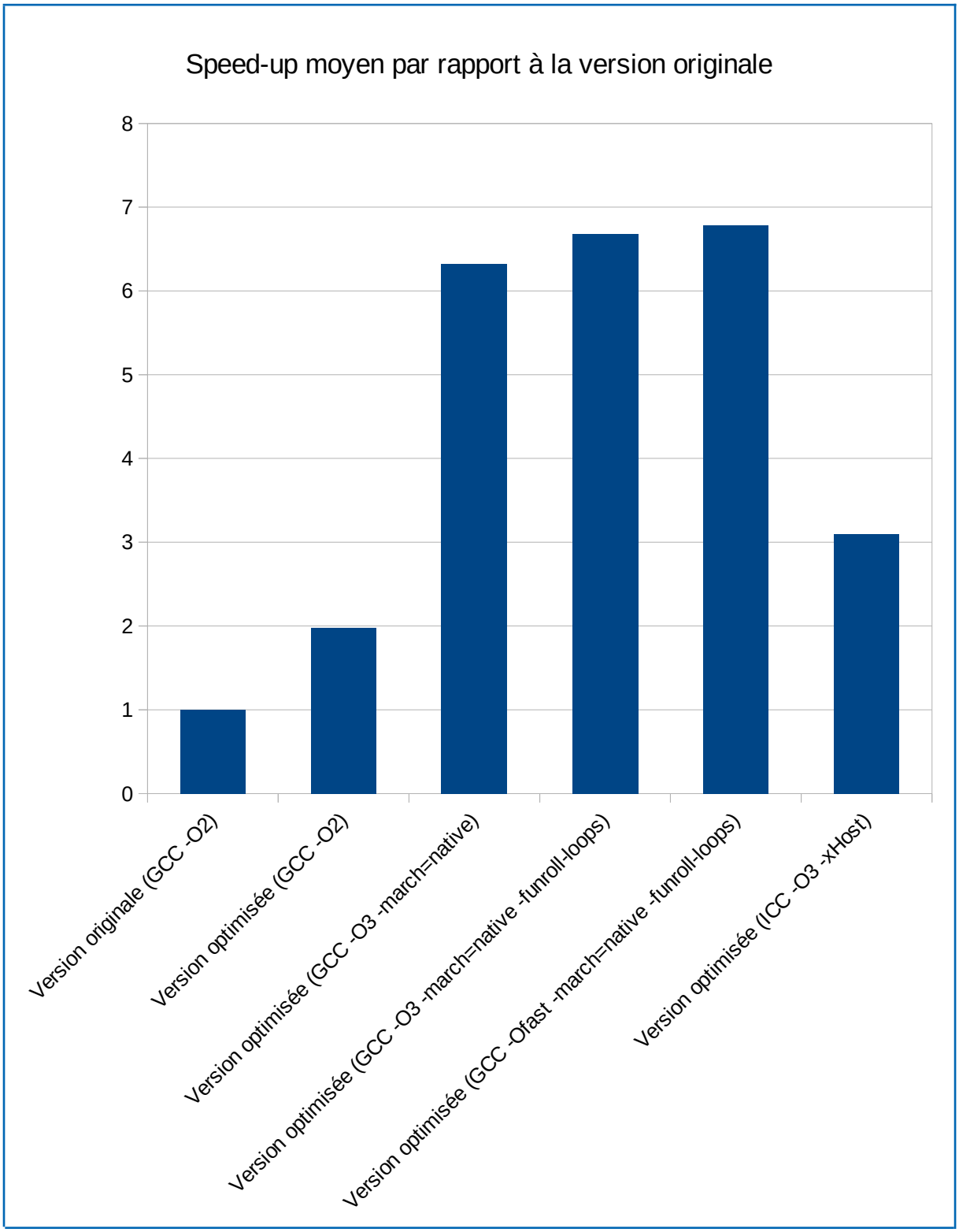


On remarque que les deux versions les plus efficaces sont :

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

Ces résultats coïncident avec ceux de la phase 1, puisque déjà, ces deux degrés étaient les plus performants avec le code original.

On peut également comparer les speed-ups, pour accentuer le résultat précédent :


Speed-up moyen par rapport à la version originale
8
7
6
5
4
3
2
1
0
Version originale (GCC -O2)
Version optimisée (GCC -O2)
Version optimisée (GCC -O3 -march=native)
Version optimisée (GCC -O3 -march=native -funroll-loops)
Version optimisée (GCC -Ofast -march=native -funroll-loops)
Version optimisée (ICC -O3 -xHost)

3.3 Parallélisation avec OpenMP

3.3.1 Code source

Il peut également être intéressant de paralléliser le noyau. Une des manières les plus simples de réaliser cela est d'utiliser l'API OpenMP.
Cette API permet de paralléliser des portions de codes, à l'aide de directives (pragmas).

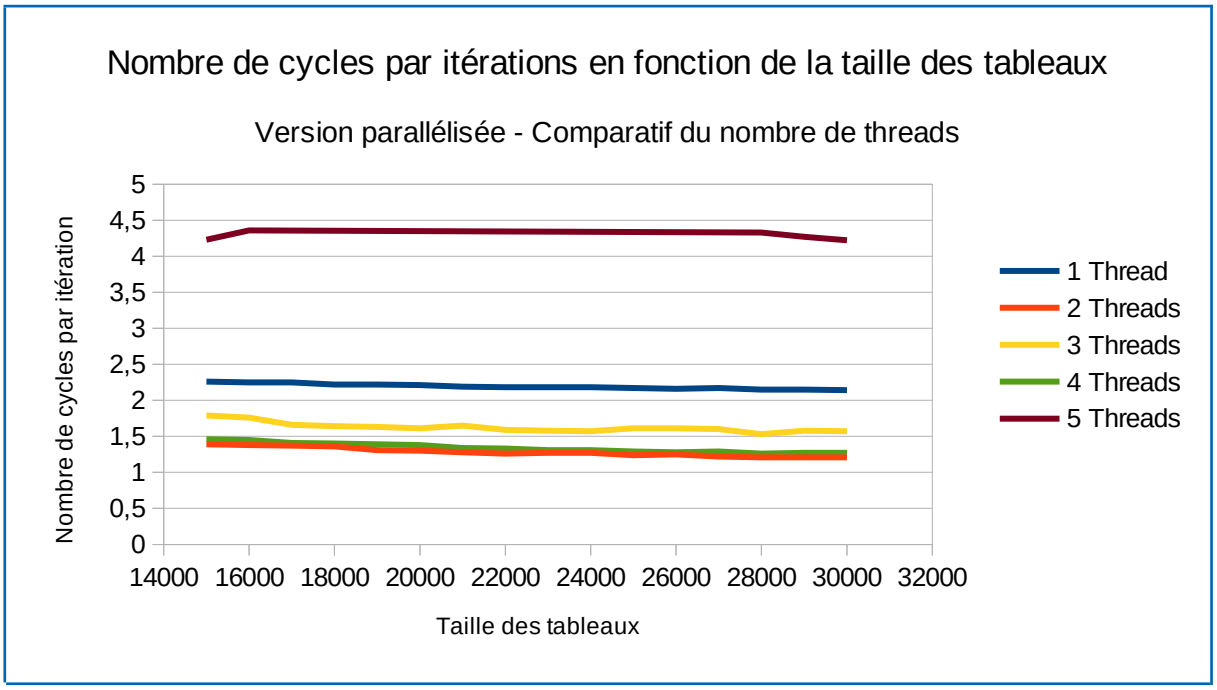
Dans notre cas, la version optimisée de notre noyau est assez simple à paralléliser, puisqu'elle ne possède que deux boucles for successives.

```
/* 4ème version : parallélisée avec OpenMP */
#include <omp.h>  // On inclut la librairie OpenMP
void baseline (int n , float a[n], float b[n], float x) {
    int i;
    #pragma omp parallel for
    for (i = 0; i < n/2; i++) {
        b[i] = (a[i] > x ? a[i] : x);
        b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
    }

    #pragma omp parallel for
    for (i = n/2; i < n; i++){
        b[i] = a[i] + x;
        b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
    }
}
```

3.3.2 Choix du nombre de threads

Concernant le compilateur, nous avons ici repris notre compilateur de base (gcc -O2).
Une des particularités de la parallélisation réside dans le nombre de threads à utiliser. Pour ce faire, nous avons utilisé le script mentionné en première partie, qui a exécuté le programme pour 1 à 5 threads.

On a ainsi obtenu la courbe suivante :



On remarque que le temps d'exécution du programme est optimal pour 2 threads. En effet, au-delà de 2 threads, le programme s'exécute plus lentement, du fait d'un phénomène appelé *overhead*.

En effet, partitionner une tâche entre un nombre élevé de threads donne à chaque thread trop peu de travail pour que le temps système nécessaire au démarrage et à la terminaison des threads surcharge le travail utile.
De plus, cela augmente le coût de la synchronisation entre les threads.

Afin de ne pas voir se produire ce phénomène, il est donc important de choisir judicieusement le nombre de threads, par exemple via une courbe comme ci-dessus.
Néanmoins, cette limite est différente sur chaque machine puisqu'elle dépend des ressources matérielles (i.e. du processeur).

Pour la suite, nous utiliserons donc deux threads.

3.3.3 Mesures

Pour n = 25000, et 2 threads on a :

Moyenne (cycles/itérations)	1,24
Minimum (cycles/itérations)	1,21
Maximum (cycles/itérations)	1,27
Médiane (cycles/itérations)	1,25
Écart-type (cycles/itérations)	0,02

On remarque dans un premier temps sont légèrement moins régulières que celles mesurées précédemment. Ceci est certainement dû à l'utilisation de la librairie OpenMP.

Si l'on compare à la version optimisée (sans parallélisation), compilée avec gcc -O2 (compilateur de base), on a :

$$G_{moyen}(V_{OMP}(gcc-O2)/V_{OPT\,3}(gcc-O2)) \approx \frac{2.03}{1.24} \approx 1.64$$

La version parallélisée est donc 1.64 fois plus rapide que la version optimisée non-parallélisée. Le gain de performances est donc non-négligeable.

4 Travaux sur le cache L3 – Étienne SAUVÉE

4.1 Optimisations du code source

Nous allons suivre un processus itératif afin d'améliorer le speed-up de la version originale. Nous prenons des tableaux de taille d'environ 200000 pour rentrer dans le cache L3.

4.1.1 Version originale

```
/* Original */
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;
        }
}

#endif
```

Voici la version originale du code, nous exécutons Maqao afin d'avoir une base sur les différentes optimisations à venir.

Global Metrics

Total Time (s)		137.67
Time in loops (%)		99.9
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	7.94
	Nb Loops to get 80%	2

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 6	63-69	baseline:kernel.c	baseline	52.56
Loop 5	63-74	baseline:kernel.c	baseline	47.31
Loop 3	13-14	baseline:driver.c	main	0.01
Loop 0	51-52	baseline:driver.c	main	0.01
Loop 4	13-14	baseline:driver.c	main	0.01

Nous observons que le programme passe la majorité du temps dans deux boucles, celles présentes dans la fonction baseline. Maqao préconise l'utilisation de l'option de compilation -funroll-loops.
Maqao nous indique aussi les optimisations possible sur ces deux boucles : cette boucle peut être déroulée, et nous pouvons enlever les expressions conditionnelles en les plaçant par exemple en dehors de la boucle.

Coverage 52.56 %
Function baseline
Source file and lines kernel.c:63-69
Module baseline
The loop is defined in /home/esauvee/Desktop/AOA/kernel.c:63-69.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

4.1.2 Optimisation 1

Nous allons donc tout d'abord à sortir au maximum les conditions des boucles (comme nous recommandait maqao). Nous ne pouvons pas sortir toutes les conditions mais nous pouvons séparer la première boucle en 2 afin d'enlever un else if du programme basique. Cela donne :

```
#ifdef OPT1
/* 1ère optimisation - If Hoisting */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                if (a[i] > x)
                        b[i] = a[i];
                else
                        b[i] = x;
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
        }

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

Exécutons maintenant Maqao sur ce nouveau programme :

Global Metrics

Total Time (s)		125.3
Time in loops (%)		99.92
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.53
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	7.96
	Nb Loops to get 80%	3

Nous pouvons dans un premier temps remarquer que nous gagnons en complexité de flux, en passant de 2 à 1,53

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 5	16-18	baseline:kernel.c	baseline	52.86
Loop 6	12-13	baseline:kernel.c	baseline	23.73
Loop 7	5-9	baseline:kernel.c	baseline	23.3
Loop 0	51-52	baseline:driver.c	main	0.02
Loop 4	13-14	baseline:driver.c	main	0.01
Loop 3	13-14	baseline:driver.c	main	0

Nous avons une boucle en plus, développons par exemple la première afin de voir se que nous informe Maqao.

Coverage	52.86 %
Function	baseline
Source file and lines	kernel.c:16-18
Module	baseline

The loop is defined in /home/esauvee/Desktop/AOA/kernel.c:16-18.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
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

La première boucle pourrait être encore améliorée avec du hoisting. Pour les deux autres boucles ayant une couverture importante dans le programme, Maqao nous informe juste de la possibilité du déroulage.

4.1.3 Optimisation 2

l'optimisation numéro 2 consiste à ajouter un loop-fusion à notre première optimisation, afin de rassembler les deux boucles présentes dans le programme original en une seul. Nous aurons donc ici deux boucles, ayant déjà séparé la première boucle en deux pour réduire le nombre d'expressions conditionnelles.

```
/* 2ème optimisation - If Hoisting & Loop fusion */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                if (a[i] > x)
                        b[i] = a[i];
                else
                        b[i] = x;

                if(b[i] < 0.0)
                        b[i] = 0.0;
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;

                if(b[i] < 0.0)
                        b[i] = 0.0;
        }
}
```

Exécutons maintenant Maqao.

Global Metrics

Total Time (s)		101.12
Time in loops (%)		99.86
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		2.71
Array Access Efficiency (%)		71.43
Clean	Potential Speedup	1.07
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.07
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.92
	Nb Loops to get 80%	2

Nous remarquons que la complexité du flux à augmenté et que l'efficacité de l'accès aux tableaux à diminué.

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 6	26-33	baseline:kernel.c	baseline	57.07
Loop 5	36-40	baseline:kernel.c	baseline	42.77
Loop 0	51-52	baseline:driver.c	main	0.02
Loop 3	13-14	baseline:driver.c	main	0
Loop 4	13-14	baseline:driver.c	main	0

Nous avons donc maintenant seulement deux boucles avec la fusion.

Coverage	57.07 %
Function	baseline
Source file and lines	kernel.c:26-33
Module	baseline

The loop is defined in /home/esauvee/Desktop/AOA/kernel.c:26-33.

The related source loop is not unrolled or unrolled with no peel/tail loop.
This loop has 4 execution paths.

The presence of multiple execution paths is typically the main/first bottleneck.
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

Ex: if (x<0) x=0 => x = (x<0 ? 0 : x) (or MAX(0,x) after defining the corresponding macro)

Ici encore, nous sommes informés de la possibilité de dérouler les boucles, et que du hoisting en remplaçant les expressions conditionnelles est possible.

4.1.4 Optimisation 3

L'optimisation 3 consiste à remplacer les expressions conditionnelles présentes dans nos deux boucles.

```
/* 3ème optimisation - Remplacement des expressions conditionnelles */
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        for (i = 0; i < n/2; i++) {
                b[i] = (a[i] > x ? a[i] : x);
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }

        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }
}
```

Global Metrics

Total Time (s)		84.9
Time in loops (%)		99.88
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		75.01
Clean	Potential Speedup	1.17
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.10
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.93
	Nb Loops to get 80%	2

Nous arrivons ici à une complexité du flux de 1.

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 5	53-55	baseline:kernel.c	baseline	63.78
Loop 6	48-50	baseline:kernel.c	baseline	36.07
Loop 3	13-14	baseline:driver.c	main	0.01
Loop 4	13-14	baseline:driver.c	main	0.01
Loop 0	51-52	baseline:driver.c	main	0.01

En analysant les deux boucles principales, on remarque que Maqao ne nous recommande plus que d'utiliser l'option -funroll-loops pour optimiser.

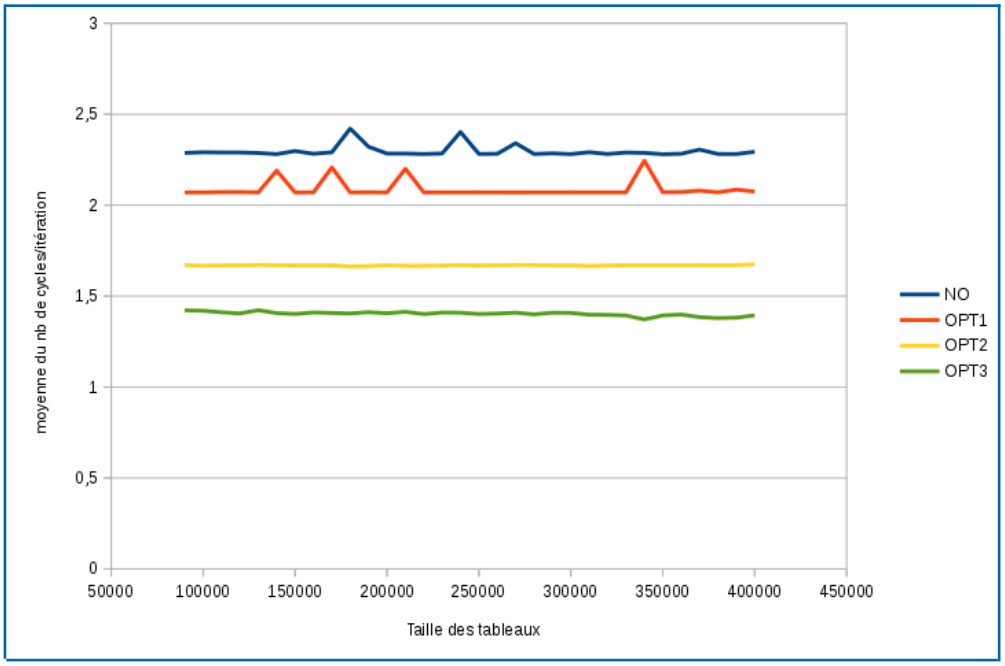
Coverage	36.07 %
Function	baseline
Source file and lines	kernel.c:48-50
Module	baseline

The loop is defined in /home/esauvee/Desktop/AOA/kernel.c:48-50.

The related source loop is not unrolled or unrolled with no peel/tail loop.

4.1.5 Bilan

Avant de finir notre comparaison, nous allons tracer un graphe représentant la version de base avec les 3 optimisations. Le graphe représente la moyenne des cycles par itérations en fonction de la taille des tableaux.



Ce résultat nous confirme le bon déroulement de la démarche itérative suivie, chaque étape (optimisation) nous fait gagner en temps. Nous garderons donc pour la suite l'optimisation 3, que nous ne pouvons améliorer beaucoup plus en modifiant le code source.

4.2 Utilisation d'optimisations plus agressives

On va maintenant modifier la compilations de base (GCC -O2) avec notre code source optimisé (l'optimisation 3).

Au sein de la première phase du projet, nous avions pu remarquer que l'optimisation icc -O3 -xHost, avec le code-source de base, produisait le meilleur résultat en termes cycles par itérations sur ma machine, contrairement aux autres machines qui produisaient de meilleurs résultats avec gcc -O3 -march=native -funroll-loops.
Nous considérer ici uniquement 4 compilations efficaces lors de la phase 1 sur cette machine, à savoir:

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

4.2.1 ICC -O3 -xHost

En compilant avec icc -O3 -xHost, on obtient les résultats suivants via Maqao :

Global Metrics

Total Time (s)		91.01
Time in loops (%)		99.8
Compilation Options		OK
Flow Complexity		1.00
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	2.72
	Nb Loops to get 80%	1

Nous observons que la vectorisation n'est pas totale, un speed-up de 2,72 est encore atteignable.

Comparons les moyennes de cycles/itération par rapport à la version optimisée compilée avec gcc -O2.

Taille tableaux	gcc -O2	icc -O3 -xHost	Speed-up		Speed-up moyen:
90000	1,42	0,78	1,82		1,82
100000	1,42	0,75	1,90		
110000	1,41	0,78	1,81		
120000	1,40	0,76	1,85		
130000	1,42	0,78	1,82		
140000	1,41	0,75	1,88		
150000	1,40	0,78	1,80		
160000	1,41	0,76	1,86		
170000	1,41	0,80	1,75		
180000	1,40	0,75	1,86		
190000	1,41	0,78	1,81		
200000	1,41	0,75	1,86		
210000	1,41	0,78	1,81		
220000	1,40	0,75	1,87		
230000	1,41	0,78	1,81		
240000	1,41	0,76	1,85		
250000	1,40	0,79	1,78		
260000	1,40	0,75	1,87		
270000	1,41	0,78	1,81		
280000	1,40	0,76	1,85		
290000	1,41	0,78	1,81		
300000	1,41	0,75	1,88		
310000	1,40	0,78	1,79		
320000	1,40	0,76	1,84		
330000	1,39	0,79	1,76		
340000	1,37	0,78	1,75		
350000	1,39	0,86	1,62		
360000	1,40	0,76	1,84		
370000	1,38	0,79	1,75		
380000	1,38	0,76	1,81		
390000	1,38	0,79	1,75		
400000	1,39	0,76	1,83		

Par rapport à la version optimisée compilée via gcc – O2, nous avons donc un speed-up moyen de 1,82.

4.2.2 GCC -O3 -march=native

En compilant avec GCC -O3 -march=native, on obtient les résultats suivants via Maqao:

Global Metrics

Total Time (s)		26.33
Time in loops (%)		99.66
Compilation Options		binary: -funroll-loops is missing.
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	1.33
	Nb Loops to get 80%	2

La encore, un speed-up est encore possible en vectorisant totalement, et Maqao recommande d'utiliser -funroll-loops.
Comparons les moyennes de cycles/itération par rapport à la version optimisée compilée avec gcc -O2.

Taille tableaux	gcc -O2	gcc -O3 -march=native	Speed-up		Speed-up moyen:
90000	1,42	0,43	3,31		3,14
100000	1,42	0,43	3,26		
110000	1,41	0,44	3,20		
120000	1,40	0,46	3,07		
130000	1,42	0,44	3,23		
140000	1,41	0,43	3,27		
150000	1,40	0,44	3,18		
160000	1,41	0,45	3,14		
170000	1,41	0,46	3,07		
180000	1,40	0,43	3,26		
190000	1,41	0,44	3,21		
200000	1,41	0,44	3,19		
210000	1,41	0,48	2,95		
220000	1,40	0,43	3,25		
230000	1,41	0,44	3,17		
240000	1,41	0,44	3,20		
250000	1,40	0,47	2,99		
260000	1,40	0,43	3,26		
270000	1,41	0,48	2,93		
280000	1,40	0,44	3,18		
290000	1,41	0,45	3,13		
300000	1,41	0,44	3,20		
310000	1,40	0,44	3,16		
320000	1,40	0,44	3,17		
330000	1,39	0,44	3,13		
340000	1,37	0,46	2,98		
350000	1,39	0,45	3,10		
360000	1,40	0,44	3,18		
370000	1,38	0,48	2,89		
380000	1,38	0,46	2,98		
390000	1,38	0,45	3,07		
400000	1,39	0,44	3,16		

Par rapport à la version optimisée compilée via gcc – O2, nous avons donc un speed-up moyen de 3,14 ce qui est meilleur qu'avec icc -O3 -xHost, tandis que durant la phase 1 cette dernière compilation fournissait de meilleurs résultats..

4.2.3 gcc_-O3-march=native-funroll-loops-g-Wall

En compilant avec gcc_-Ofast -march=native -funroll-loops, on obtient les résultats suivants via Maqao:

Global Metrics

Total Time (s)		51.72
Time in loops (%)		99.63
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		74.98
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	1.99
	Nb Loops to get 80%	2

Nous remarquons ici que l'efficacité des accès aux tableaux à diminué, mais le speed-up potentiel en vectorisant totalement aussi.

Comparons les moyennes de cycles/itération par rapport à la version optimisée compilée avec gcc -O2.

Taille tableaux	gcc -O2	gcc -Ofast -march=native -funroll-loops	Speed-up		Speed-up moyen:
90000	1,42	0,42	3,38		3,27
100000	1,42	0,41	3,45		
110000	1,41	0,42	3,36		
120000	1,40	0,48	2,93		
130000	1,42	0,42	3,37		
140000	1,41	0,42	3,35		
150000	1,40	0,42	3,34		
160000	1,41	0,44	3,20		
170000	1,41	0,43	3,26		
180000	1,40	0,42	3,34		
190000	1,41	0,42	3,36		
200000	1,41	0,43	3,27		
210000	1,41	0,47	3,00		
220000	1,40	0,42	3,33		
230000	1,41	0,42	3,35		
240000	1,41	0,42	3,35		
250000	1,40	0,46	3,07		
260000	1,40	0,42	3,34		
270000	1,41	0,42	3,35		
280000	1,40	0,42	3,33		
290000	1,41	0,44	3,20		
300000	1,41	0,42	3,34		
310000	1,40	0,42	3,32		
320000	1,40	0,42	3,32		
330000	1,39	0,44	3,20		
340000	1,37	0,43	3,19		
350000	1,39	0,42	3,32		
360000	1,40	0,42	3,33		
370000	1,38	0,43	3,22		
380000	1,38	0,44	3,13		
390000	1,38	0,43	3,21		
400000	1,39	0,43	3,24		

Par rapport à la version optimisée compilée via gcc – O2, nous avons donc un speed-up moyen de 3,27.

4.2.4 gcc_-Ofast-march=native-funroll-loops-g-Wall

En compilant avec gcc_-Ofast-march=native-funroll-loops-g-Wall, on obtient les résultats suivants via Maqao:

Global Metrics

Total Time (s)		53.1
Time in loops (%)		99.65
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		74.98
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	1.95
	Nb Loops to get 80%	2

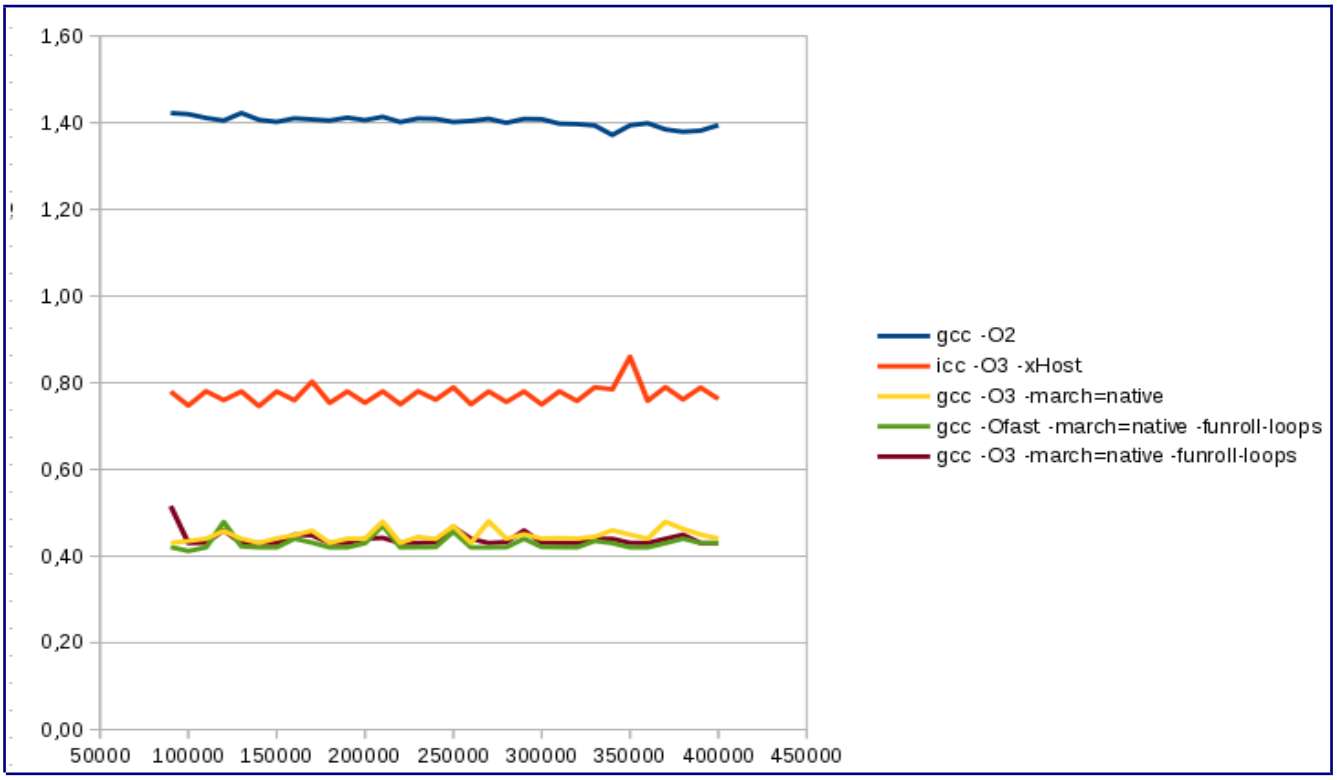
Comparons les moyennes de cycles/itération par rapport à la version optimisée compilée avec gcc -O2.

Taille tableaux	gcc -O2	gcc -O3 -march=native -funroll-loops	Speed-up		Speed-up moyen:
90000	1,42	0,52	2,76		3,19
100000	1,42	0,43	3,30		
110000	1,41	0,43	3,27		
120000	1,40	0,46	3,06		
130000	1,42	0,43	3,31		
140000	1,41	0,43	3,27		
150000	1,40	0,43	3,26		
160000	1,41	0,45	3,14		
170000	1,41	0,45	3,14		
180000	1,40	0,43	3,26		
190000	1,41	0,43	3,28		
200000	1,41	0,44	3,20		
210000	1,41	0,44	3,20		
220000	1,40	0,43	3,26		
230000	1,41	0,43	3,28		
240000	1,41	0,43	3,26		
250000	1,40	0,47	3,00		
260000	1,40	0,44	3,19		
270000	1,41	0,43	3,27		
280000	1,40	0,43	3,24		
290000	1,41	0,46	3,07		
300000	1,41	0,43	3,26		
310000	1,40	0,43	3,25		
320000	1,40	0,43	3,25		
330000	1,39	0,44	3,17		
340000	1,37	0,44	3,12		
350000	1,39	0,43	3,24		
360000	1,40	0,43	3,25		
370000	1,38	0,44	3,15		
380000	1,38	0,45	3,07		
390000	1,38	0,43	3,21		
400000	1,39	0,43	3,24		

Le speed-up moyen est ici de 3,19.

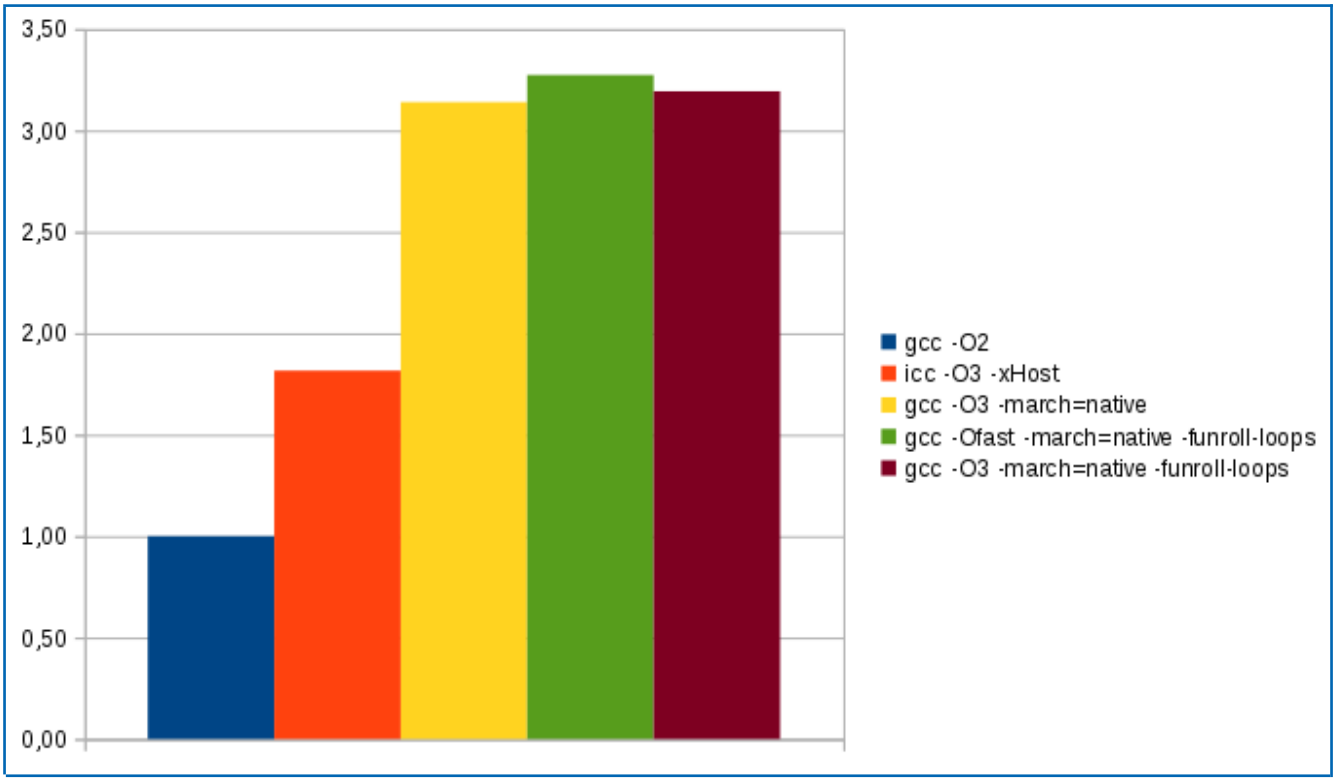
4.2.5 Bilan

Comparer les options de compilations nous amène à ce graphe:



Nous remarquons peut de différence sur cette machine pour les trois options de compilation suivante :
gcc -O3 -march=native, gcc -O3 -marche=native -funroll-loops, et gcc -Ofast -march=native -funroll-loops.

On peut également comparer les speed-ups, pour accentuer le résultat précédent :



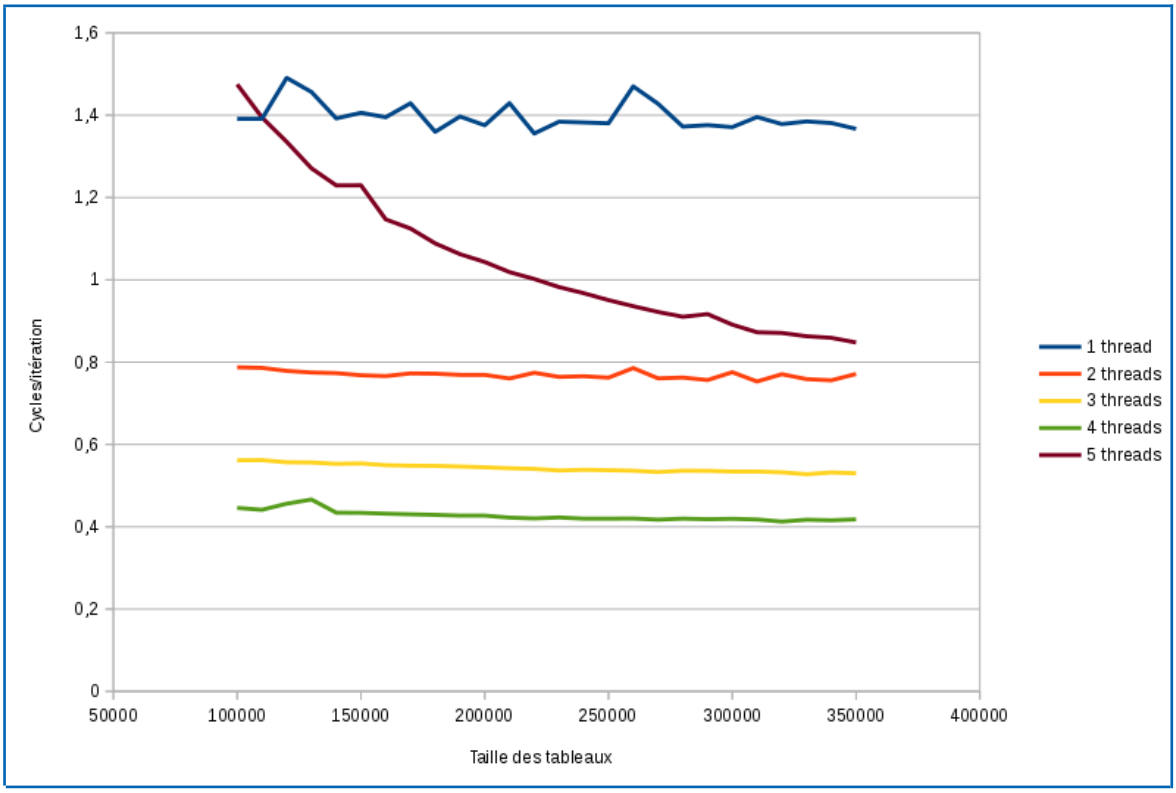
On peut ici bien voir que la plus efficace est de peu gcc -Ofast -march=native -funroll-loops, tandis que lors de la phase 1, c'était icc -O3 -xHost qui fournissait de meilleurs résultats. Ici icc produit des résultats peu intéressants, avec un speed-up inférieur à 2.

4.3 Parallélisation avec OpenMP

Nous allons pour finir l'étude d'optimisations analysé les résultats de la parallélisation de notre code optimisé pour le kernel. La compilation sera effectuée avec gcc -O2 -fopenmp.

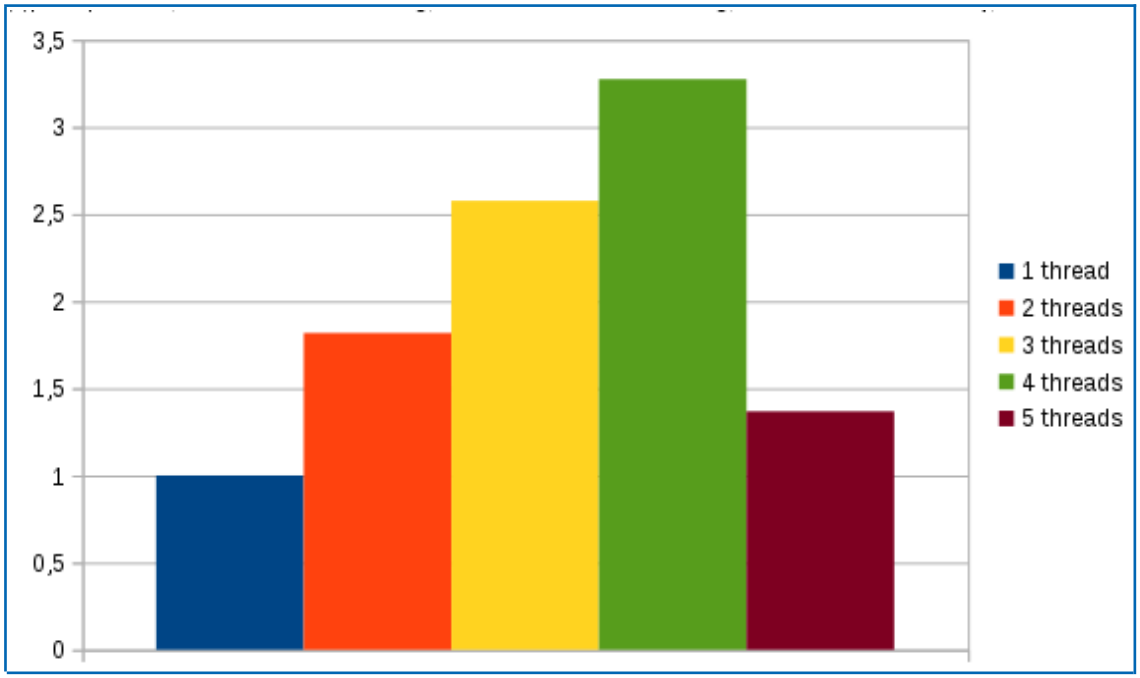
```
#include <omp.h>  // On inclut la librairie OpenMP
void baseline (int n , float a[n], float b[n], float x) {
        int i;
        #pragma omp parallel for
        for (i = 0; i < n/2; i++) {
                b[i] = (a[i] > x ? a[i] : x);
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }

        #pragma omp parallel for
        for (i = n/2; i < n; i++){
                b[i] = a[i] + x;
                b[i] = (b[i] < 0.0 ? 0.0 : b[i]);
        }
}
```

Traçons les courbes représentant les moyennes de cycle par itération en fonction de la taille des tableaux, pour un nombre de threads variant de 1 (pas de parallélisation) à 5.



Nous pouvons observer que sur cette machine, la parallélisation avec 4 threads est la plus performante, au-delà les overheads liés à l'utilisation des threads (le programme doit attendre la fin de l'exécution de tous les threads pour continuer) entrainent de moins bonnes performances.

Comparons les speed-ups moyens:



On confirme bien que l'utilisation de 4 threads sur cette machine fournira de meilleurs résultats.

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

https://en.wikipedia.org/wiki/Loop_optimization

https://en.wikipedia.org/wiki/OpenMP

https://www.ljll.math.upmc.fr/SGI/pdf/jacobi3d.pdf

https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf