Rapport Projet AOB

Analyse et Optimisation des performances d'un code

Par : Ibrahima KOUNDOUL

Encadreur : M. IBNAMAR Mohammed-Salah

Résumé du sujet

En programmation informatique, l’optimisation de code permet généralement au programme résultant de s’exécuter plus rapidement, de prendre moins de place en mémoire, de limiter sa consommation de ressources (par exemple les fichiers), ou de consommer moins d’énergie électrique.

Notre travail consiste à analyser et optimiser des performances d'un code donné qui a pour but d'appliquer des transformations sur du contenu vidéo brut (RGB) non compressé stocké dans un fichier.

Table des matières

Index des illustrations

Introduction

Ce projet s’inscrit dans le cadre de le formation scientifique et technologique à l’ISTY.

Nous allons devoir analyser et d'améliorer les performances des codes fournis en documentant et justifiant les étapes suivies.

Dans un premier temps nous allons étudier l’environnement dans lequel les traitements seront faits, notamment l’architecture et la portabilité du compilateur (sur les codes) que nous allons utiliser.

Ensuite nous procéderons aux paramétrages nécessaires afin d’avoir les calculs les plus précis.

Une deuxième phase sera l’Analyse et la mesure des performances de notre code. Dans cette partie nous allons étudier, mesurer, interpréter mais aussi comparer les métriques ou résultats obtenus avec nos différentes versions d’optimisation des codes.

Ensuite nous allons utiliser un autre compilateur et flags de compilation et les tester sur nos différentes versions d’optimisations eu précédemment et comparer les résultats et voir si le code a été optimisé ou pas.

Ensuite nous finirons par une brève synthèse du travail effectué, nous en ressortiront les grandes lignes et enfin nous terminerons par un conclusion qui sera un résumé de ce projet, nos impressions.

Vous trouverez sur la dernière page du rapport l’ensemble des documents, ou liens sources sur lesquels nous nous sommes appuyés pour réaliser ce travail.

BONNE LECTURE !!!!

Analyse et Mesures des Performances du code original avec GCC

Environnement

- **Architecture**: x86_64
- **CPU name:** Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz
- **CPU stepping** : 3
- **CPU max MHz**: 3300,0000
- **CPU min MHz**: 800,0000
- **Cache L1** : 32KB
- **Cache L2** : 256 KB
- **Cache L3** : 3073 KB
- **Ram** : 4GB

L'analyse effectué sur le code original, c'est à dire sans optimisation de notre part a été effectué sur un processeur Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz, Le système d'exploitation est un Ubuntu en 64 bits installé en Dual Boot (linux natif) ce qui facilite grandement les mesures de performances et permet l'utilisation de Likwid.

Les différentes valeurs des caches ont été trouvées grâce à la commande **lscpu,** on pouvait aussi utiliser la commande **likwid-topology -g**.

Notre code original est compilé avec le compilateur gcc et les flags suivantes : O1,O2,O3, Ofas**t**

Portabilité de gcc

Par défaut, GCC compile du code pour le même type de machine que celui que vous utilisez.

La version de notre compilateur est ***gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0*** compatible sur ***l'architecture ADM64 (aka x86_64)*** dont nous disposons.

Les options O1, O2, O3, Ofast contrôle différentes sortes d'optimisations :

-O1
La compilation optimisante prend un peu plus de temps, et beaucoup plus de mémoire pour une grande fonction.

- **Sans -O**

Le but du compilateur est de réduire le coût dû à la compilation et de permettre au débogage de produire les résultats escomptés. Les instructions sont indépendantes : si on stoppe le programme sur un point d'arrêt (breakpoint) entre des instructions, on peut ensuite attribuer une nouvelle valeur à n'importe quelle variable ou modifier l'indicateur d'instruction courante (program counter) pour le faire pointer vers n'importe quelle autre instruction de la fonction et obtenir exactement les mêmes résultats que ceux qu'on pourrez escompter en partant du code source.

- **Avec -O**

Le compilateur essaie de réduire la taille du code et le temps d'exécution, sans effectuer d'optimisation nécessitant beaucoup de temps de compilation.

-O2

Optimiser encore plus. GCC effectue pratiquement toutes les optimisations supportées qui n'impliquent pas un compromis espace/vitesse. Le compilateur n'effectue pas de déroulement des boucles ou de mise en ligne de fonctions quand on spécifiee **-O2**. Comparée à **-O**, cette option augmente à la fois le temps de compilation et la performance du code généré.

Il active toutes les optimisations optionnelles à l'exception du déroulement des boucles, de la mise en ligne de fonctions, et du renommage de registres. Il active également l'option **-fforce-mem(permet de produire un meilleur code en faisant de toutes les références mémoire des sous-expressions potentiellement communes)** sur toutes les machines, et élimine le pointeur de cadre sur les machines pour lesquelles faire de la sorte n'interfère pas avec le débogage.

-O3

Optimiser toujours plus. **-O3** active toutes les optimisations spécifiées par **-O2** et l'option **-frename-registers,** avec cette option on essaye d'éviter les fausses dépendances dans le code ordonnancé en utilisant des libres après l'allocation des registres. Cette optimisation bénéficiera le plus aux processeurs possédant beaucoup de registres.

-Ofast active toutes les optimisations «-O3». Il permet également des optimisations qui ne sont pas valides pour tous les programmes conformes aux normes

Analyse des métriques et performance du code original

Context:

Nous disposons en entrée d'une vidéo avec des algorithmes de traitement d'images destiné à être entré dans un réseau de neurones. Plusieurs analyses sont effectuées sur notre entrée :

1. **Color inversion Fichier invert.c :**

   la fonction process_baseline() nous permet d'inverser les couleurs (color inversion) qui nous permet de créer un mask négatif utilisé pour l'analyse de profondeur afin d'approximer les dimensions en 3D, des objets détectés .

2. **La détection de contours Fichier sobel.c :**

   Afin d'effectuer l'analyse de contours avec l'algorithme de Sobel, l'image doit être convertie en niveau de gris.afin d'éviter les artefacts dus à la variation de couleurs (contrast).

   Nous disposons de 2 fonctions de conversion:

   La fonction void grayscale_weighted(u8 *frame) :

   qui se base sur la pondération des composantes RGB (Rouge,Green,, le principe est de mélanger différentes quantités de R, G, B pour avoir une variante de gris

   La fonction void grayscale_sampled(u8 *frame) :

   qui utilise l'échantillonnage.

La différence entre les deux algorithmes réside principalement dans la qualité de l'image après conversion. **La fonction void grayscale_weighted(u8 *frame) donne les meilleurs résultats.**

Paramétrages & Mesures des Performances

hardware limits: 800 MHz - 3.30 GHz

Avant l'exécution du code on s'est assuré que notre ordinateur est bien sous secteur, cela évite les ajustements dynamiques de la fréquence du processeur aussi nous nous sommes assurer que la fréquence du processeur est fixe au maximum durant les mesures en le spécifiant manuellement avec la commande : **cpupower -c 0-3 frequency-set -u 3.3GHz**

Nous allons maintenant exécuter notre code et étudier les métriques obtenues avec les différents flags résumés dans le tableau ci-dessous après l'illustration suivante :

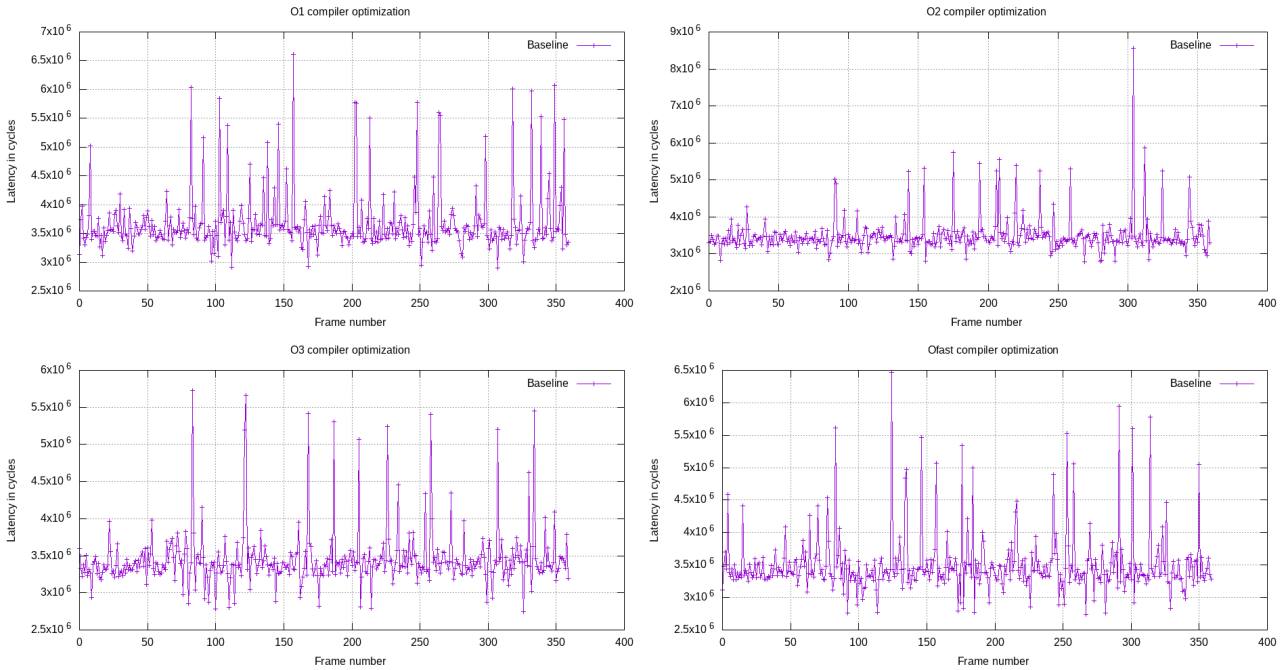

Illustration 1: Courbes de la latence dans le cycle en fonction du nombre de frames de la version sans optimisation selon les différents flags

Flags	***Cycle min***	***Cycle max***	***Average des cycles***	***Moyenne arithmétiques des cycles***	***Nombre octets par cycle***	***Déviation standard***
-O1	*2905792*	*6613216*	*4759504*	*3680895,125*	*0,751*	*14,808 %*
-O2	*2788740*	*8567512*	*5678126*	*3502557,036*	*0,789*	*14,645 %*
-O3	*2749089*	*5729299*	*4239194*	*3454464,203*	*0,800*	*11,553 %*
-Ofast	*2742024*	*6466697*	*4604360,5*	*3503488,406*	*0,789*	*13,754 %*

Tableau récapitulatif des métriques obtenus avec la version de base sous gcc

Interprétations des résultats

Nous disposons de deux métriques: cycles CPU écoulés (mesure) et les octets par cycle (taille des données / cycles écoulés --> combien de données mémoire ont été traités par cycle CPU).

A l'exécution nous observons que le flag -O3 donnaient les meilleurs résultats, il traite plus de données mémoire par cycle CPU, sollicite moins le processeur (voir colonne Average), les données variées moins.

Suivis de -O2 et -Ofast offraient presque les mêmes performances presque sauf que -Ofast sollicitait moins le processus et les données variées moins, et le nombre max de cycle CPU était très inférieur à celui de -O2, cependant ils ont traité le même nombre de données mémoire par cycle CPU(0,789).

-O1 a donné les plus mauvais.

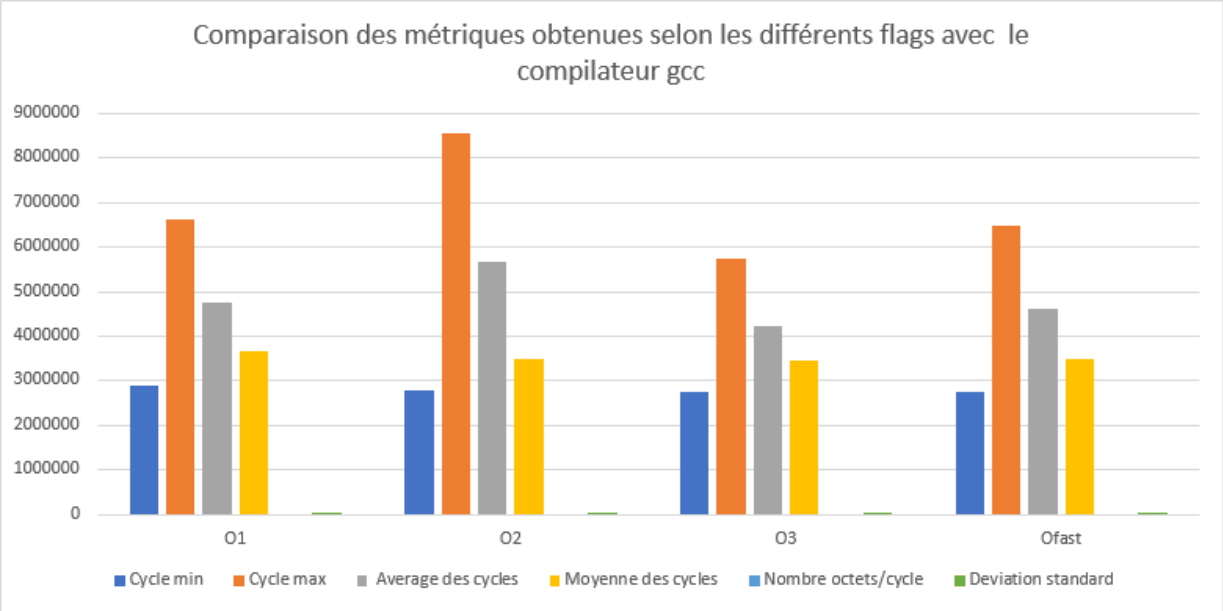

Illustration 2: Graphes comparaison des métriques obtenues selon les différents flags avec le compilateur gcc

Étude avec likwid

likwid-perfctr: compte les événements de performances matérielles.

Flags	*Cycle min*	*Cycle max*	*Average des cycles*	*Moyenne arithmétiques des cycles*	*Nombre octets par cycle*	*Déviation standard*
-O1	*2788237*	*9440324*	*6114280.5*	*4187079.708*	*0.660*	*34.386 %*
-O2	*2668524*	*26729360*	*14698942*	*3983069.311*	*0.694*	*47.896 %*
-O3	*2709625*	*10512120*	*6610872.5*	*4205004.878*	*0.658*	*38.526 %*
-Ofast	*2751147*	*10361383*	*6556265*	*3947348.519*	*0.700*	*34.113 %*

Tableau récapitulatif des métriques obtenus avec la version de base sous gcc avec likwid

Metric	Core 0	Core 1	Core 2	Core 3
Runtime (RDTSC) [s]	1.9237	1.9237	1.9237	1.9237
Runtime unhalted [s]	0.0478	1.4447	0.0808	0.0030
Clock [MHz]	3085.7096	2970.1414	3252.4370	2840.3189
CPI	0.4872	0.4118	0.5303	2.1129
Branch rate	0.1871	0.0698	0.2021	0.2010
Branch misprediction rate	0.0023	0.0010	0.0034	0.0100
Branch misprediction ratio	0.0123	0.0148	0.0166	0.0498
Instructions per branch	5.3454	14.3238	4.9482	4.9748

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	7.6948	1.9237	1.9237	1.9237
Runtime unhalted [s] STAT	1.5763	0.0030	1.4447	0.3941
Clock [MHz] STAT	12148.6069	2840.3189	3252.4370	3037.1517
CPI STAT	3.5422	0.4118	2.1129	0.8855
Branch rate STAT	0.6600	0.0698	0.2021	0.1650
Branch misprediction rate STAT	0.0167	0.0010	0.0100	0.0042
Branch misprediction ratio STAT	0.0935	0.0123	0.0498	0.0234
Instructions per branch STAT	29.5922	4.9482	14.3238	7.3980

Cette fois-ci sur likwid c'était -Ofast traitait le plus de données en mémoire et il offrait les meilleurs performances sur toutes les métriques, ensuite s'en suivait -O2 en terme de données traitées en mémoire mais présentait une déviation standard sur les données le plus élevée ensuite -O1 et celle qui offrait les performances les plus mauvaises étaient -O3.

Comment pouvons-nous expliquer cette différence entre les résultats obtenus précédemment?

Sur Linux les logiciels ont toujours une disponibilité très limitée. Par conséquent, le flux de travail expérimental sur ces ordinateurs doit être léger et avec des dépendances minimales.
Dans un tel contexte, un malentendu ou une inexactitude vrai semblablement mineurs sur certains paramètres à petite échelle peut entraîner un comportement totalement différent au niveau

macroscopique. Cela nous montre à quel point le biais de mesure peut être imprévisible et inattendu. L'utilisation de drapeaux de compilation différents et un léger changement de l'ordre de liaison lorsqu'on mesure les performances d'une application peuvent produire des résultats différents. cliquez pour voir la source

Nous allons maintenant faire une étude avec l'application maqao qui permet l'analyse statique de codes x86. Dans cette analyse on pourra détecté ce qui provoque les ralentissements, comment on pourrait réduire le coût des itérations et comment gagner en performances.

Etude avec maqao sur la fonction process_baseline (invert.c)

Grâce aux commandes suivantes nous avons pu effectuer notre analyse :

maqao cqa _inver fct-loops=process_baseline conf=expert uarch=SKYLAKE

maqao cqa _inver fct-loops=process_baseline conf=gain uarch=SKYLAKE

L'étude de la fonction process_baseline avec maqao nous a révélé que :

-La boucle source associée n'est pas déroulée, chaque itération de la boucle binaire prend 3,50 cycles. À ce niveau là: 1% de la performance de charge maximale 2% des performances maximales du magasin sont atteintes (0,86 sur 32,00 octets stockés par cycle (Go / s à 1 GHz)), Les performances sont limitées par le débit des instructions.
La boucle n'est pas vectorisée, seulement 3 % de la longueur du registre est utilisée.
En supprimant tous ces goulots d'étranglement, on pourrait réduire le coût d'une itération de 3,50 à 3,00 cycles (accélération de 1,17x).

Comme solution maqao nous propose :
de dérouler notre boucle ou en recompilant avec -funroll-loops, changer de compilateur afin de permettre la vectorisation,supprimer les dépendances entre les itérations de notre boucle

Etude avec maqao sur la fonction sobel_baseline (sobel.c)

En changeant dans le fichier run_invert.c les variantes du code par sob_baseline et executant cette fois ci avec le binaire sobel on a pu effectuer nos analyses sur la fonction sobel_baseline.

Nous avons obtenus les résultats suivants :

Flags	*Cycle min*	*Cycle max*	*Average des cycles*	*Moyenne arithmétiques des cycles*	*Nombre octets par cycle*	*Déviation standard*
-O1	*139450558*	*203336074*	*171393316*	*145568941.442*	*0.038*	*4.738 %*
-O2	*152383391*	*176478037*	*164430714*	*157960716.061*	*0.035*	*2.664 %*
-O3	*34557913*	*56276325*	*45417119*	*37502012.519*	*0.147*	*11.875 %*
-Ofast	*24230217*	*42456881*	*33343549*	*27143835.450*	*0.204*	*15.616%*

Tableau récapitulatif des métriques obtenues avec maqao de la version de base gcc

Résultats

Nous avons remarque que c'était Ofast qui donnait les meilleurs resultats en terme d'utilisation des ressources et de traitement nombre de données traitées par cycle suivi de O3. Mais ce qui attire notre attention était que le fait la fonction sobel_baseline bouffe plus de ressource que la fonction process_baseline. Voir le graphe ci-dessous (comparaison des resultats des meilleurs flags pour ces deux fonction.

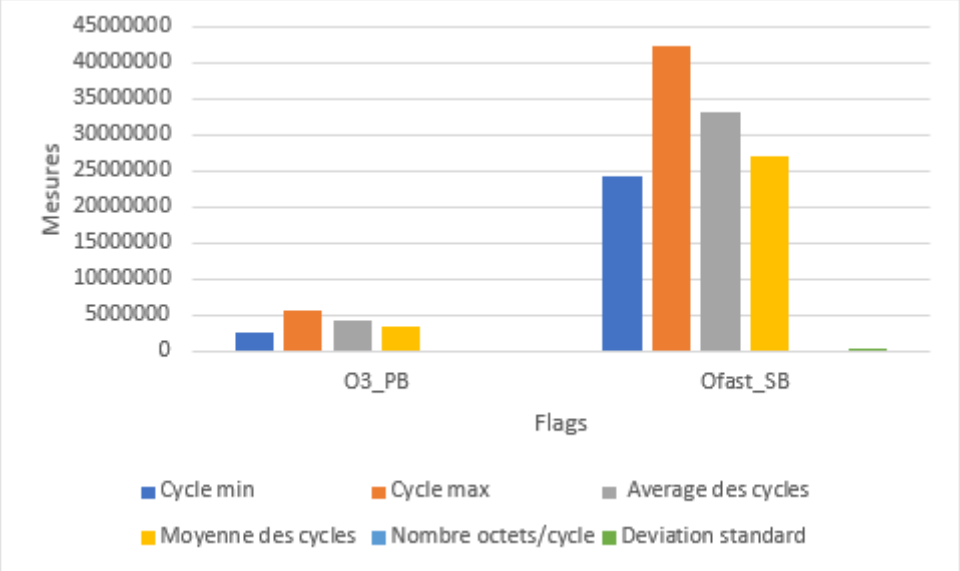

Illustration 3: graphes comparaison des résultats des meilleurs flags pour ces les fonctions process_baseline(PB) et sobel_baseline(SB).

Nous allons nous aider d'une analyse avec maqao pour mieux comprendre.
Grâce aux commandes suivantes nous avons pu effectuer notre analyse :

maqao cqa _sobel fct-loops=sobel_baseline conf=expert uarch=SKYLAKE

maqao cqa _sobel fct-loops=sobel_baseline conf=gain uarch=SKYLAKE

Résultats maqao :

Le rapport maqao nous a révélé que :

La boucle est vectorisé mais utilise une longueur de registre de 43%, c'est la boucle principale de la boucle source associée qui est déroulée par 16 (y compris la vectorisation).
0% des performances de calcul maximales sont utilisées, le code assembleur généré est beaucoup plus long parce que si on regarde à l'intérieur de la fonction sobel_baseline on appel 2X la fonction convolve_baseline ce qui nous plusieurs itérations, chaque itération de la boucle binaire prend 118,00 cycles. À ce niveau là: 11% des performances de charge de pointe sont atteintes, 5% des performances maximales du magasin sont atteintes.

Comme solution maqao nous propose :

En vectorisant entièrement notre boucle, on pourrait réduire le coût d'une itération de 118,00 à 48,00 cycles (accélération 2,46x), de recompiler avec march = skylake, d'alignez nos tableaux sur des limites de 32 octets: remplacez {void * p = malloc (taille); } avec {void * p; posix_memalign (& p, 32, taille); }.

Nous allons maintenant passé à l'optimisation de ces deux fonctions et voir comment les résultats vont évoluer.

OPTIMISATION 1 : loop interchange

l'échange de boucles est le processus d'échange de l'ordre de deux variables d'itération utilisées par une boucle imbriquée. La variable utilisée dans la boucle interne passe à la boucle externe et vice versa. Il est souvent fait pour s'assurer que les éléments d'un tableau multidimensionnel sont accessibles dans l'ordre dans lequel ils sont présents en mémoire, améliorant ainsi la localité de référence.

Nous allons appliquer cette méthode à nos 2 fonctions sobel_baseline et process_baseline et voir l'évolution des résultats :

Nous allons observer le cycles CPU écoulés (mesure) et les octets par cycle traités et les comparés aux résultats qu'on avait précédemment :

En **noir** les résultats d'avant et en **bleu** les résultats obtenus avec OPT1

Flags Base / OPT1	*Average des cycles*	*Nombre octets par cycle*
-O1	*4759504* *12841322*	*0,751* *0.330*
-O2	*5678126* *14820651*	*0,789* *0.321*
-O3	*4239194* *13123458*	*0,800* *0.321*
-Ofast	*4604360,5* *15315465.5*	*0,789* *0.306*

Tableau comparatif des métriques obtenues entre le versions de base et OPT1

On remarque cette optimisation a au contraire eu un effet négatif sur l'exécution du code, on consomme plus en ressource et on traite moins de données par cycle.

Nous allons faire une étude avec maqao et voir ce qui a fait que nous avons perdu en performance

Etude sur maqao resultat OPT1 avec gcc

Nous avons obtenons les mêmes que pour tout à l'heure, la boucle dans process_baseline n'était pas vectorisé, d'essayer un autre compilateur afin de la vectoriser, de vectoriser entière la fonction sobel_baseline, recompiler avec skylake.

OPTIMISATION 2 : vectorisation de la boucle process_baseline

La première analyse consiste à inverser les couleurs d'un frame, la fonction process_baseline nous permet d'effectuer cette opértion, une image (ou frame) est une matrice de pixels (RGB) avec une hauteur (H ou nombre de lignes) et une largeur (W ou nombre de colonnes), pour un capteur de largeur W et de hauteur H, il faut donc un frame buffer de: **W x H x 3**

Nous allons vectoriser la boucle comme suit :

Dès lors on pourra remplacer les boucles de notre fonction par :

void process_baseline(u8 *frame) ***{*** ***for (unsigned y = 0 ; y < H ; ++y)*** ***for (unsigned x = 0 ; x < W * 3 ; x += 3)*** ***{*** ***//Invert each colour component of every pixel*** ***frame[INDEX(y, x, W * 3) + 0] = 255 - frame[INDEX(y, x, W * 3) + 0]; //Red*** ***frame[INDEX(y, x, W * 3) + 1] = 255 - frame[INDEX(y, x, W * 3) + 1]; //Green*** ***frame[INDEX(y, x, W * 3) + 2] = 255 - frame[INDEX(y, x, W * 3) + 2]; //Blue*** ***}*** ***}***	***void process_baseline(u8 *frame)*** ***{*** ***for (unsigned x = 0 ; x < H*W * 3 ; x += 3)*** ***{*** ***//Invert each colour component of every pixel*** ***frame[x + 0] = 255 - frame[x + 0]; //Red*** ***frame[x + 1] = 255 - frame[x + 1]; //Green*** ***frame[x + 2] = 255 - frame[x + 2]; //Blue*** ***}*** ***}***

A l'exécution nous obtenons les résultats suivants :

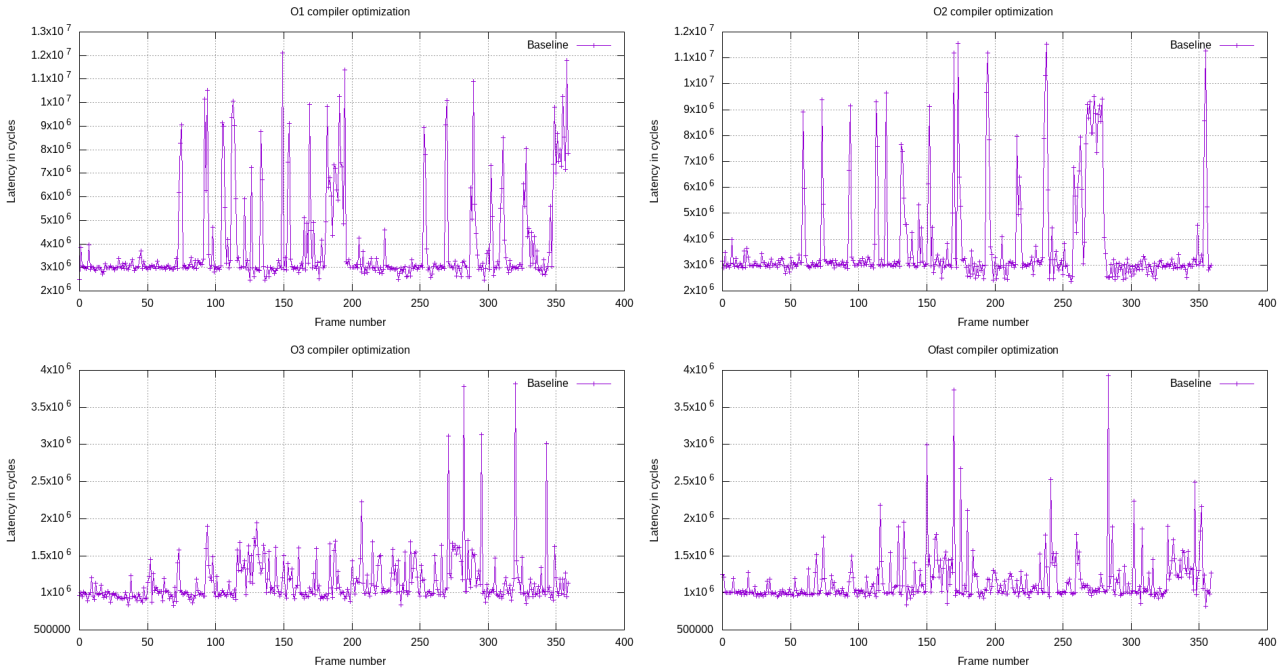

Illustration 4: Les Courbes de la latence dans le cycle en fonction du nombre de frames de la version OPT2 selon les différents flags

On remarque on a obtenu de meilleurs résultats que précédemment, surtout avec le flag -Ofast.

Nous allons comparer les métriques afin de mieux observer cette différence

Comparaison des métriques entre les performances de Base et OPT2

Nous allons maintenant comparer les métriques obtenus dans l'optimisation 2 à la version de base.

le graphe ci-dessous représente la différence de performance entre les métriques pour ces 2 versions.

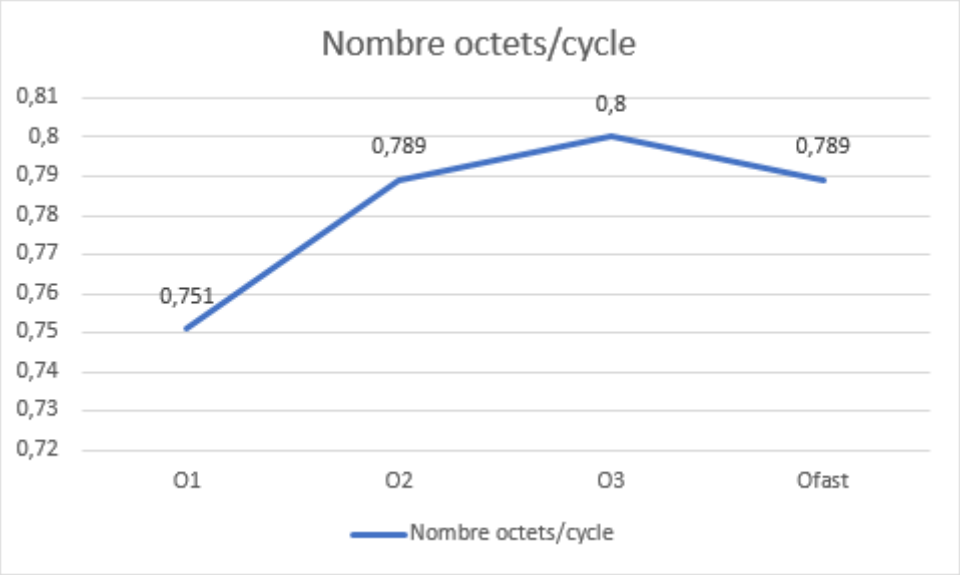

Illustration 5: Courbe du nombre d'octets/cycle traités avec la version de base avec le compilateur gcc

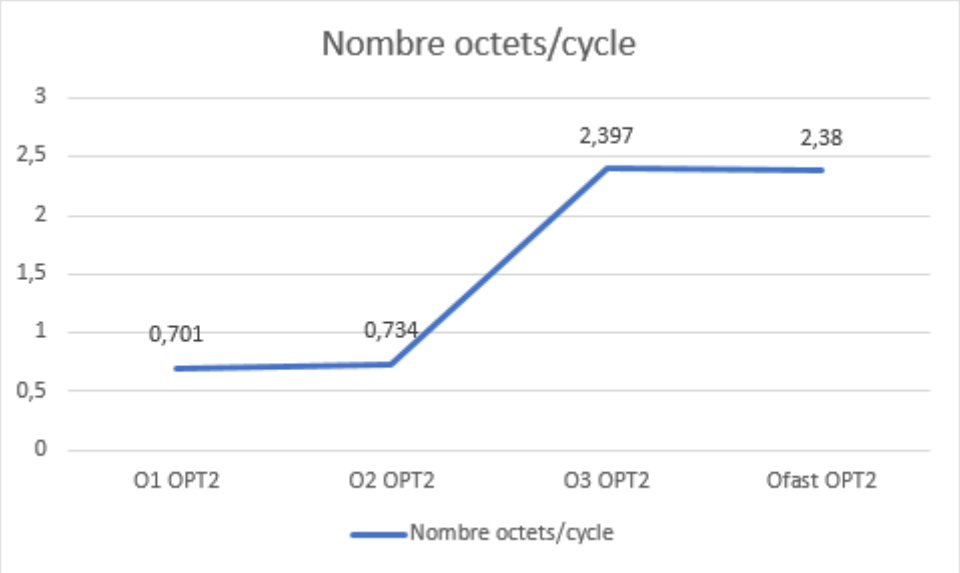

Illustration 6: Courbe du nombre d'octets/cycle traités avec la version OPT2 avec le compilateur gcc

Flags	*Cycle min*	*Cycle max*	*Average des cycles*	*Moyenne des cycles*
O1	*2905792*	*6613216*	*4759504*	*3680895,125*
O2	*2788740*	*8567512*	*5678126*	*3502557,036*
O3	*2749089*	*5729299*	*4239194*	*3454464,203*
Ofast	*2742024*	*6466697*	*4604360,5*	*3503488,406*

Flags -OPT2	*Cycle min*	*Cycle max*	*Average des cycles*	*Moyenne des cycles*
O1	*2450624*	*12097267*	*7273945,5*	*3944366,778*
O2	*2378643*	*11556857*	*6967750*	*3766272,881*
O3	*830516*	*3813663*	*2322089,5*	*1153481,603*
Ofast	*821640*	*3928040*	*2374840*	*1161512,594*

Tableau comparatif des métriques obtenues entre OPT2 et la version de base sous gcc

Interprétations des résultats :

Nous remarquons que cette optimisation a été bénéfiques pour le compilateur gcc avec les flags O3 et Ofast, on gagnait en performance et on solliciter moins les ressources, on traite plus de données par cycle (voir courbes du haut).

Nous avons ensuite effectuer une autre analyse avec maqao pour mieuxx optimiser notre solution.

Étude OPT2 sous maqao

Rappelons que nous avons appliquer notre optimisation que sur la fonction process_baseline donc la fonctions sobel_baseline grade les mêmes résultats (voir résultats plus haut) dans l'analyse avec maqao. L'analyse suivante n'a porté que sur la fonction process_baseline

Cette fois-ci la boucle est vectorisé, maqao nous dit que en vectorisant entièrement notre boucle, on pourrait réduire le coût d'une itération de 3,00 à 1,50 cycle (accélération 2,00x).
Nos performances sont limitées en écrivant des données dans des caches / RAM (l'unité de stockage est un goulot d'étranglement).
En supprimant tous ces goulots d'étranglement, on pourrait réduire le coût d'une itération de 3,00 à 2,75 cycles (accélération de 1,09x).
On pourrait arriver à cela en essayant d'écrire moins d'éléments de tableau et de fournir plus d'informations au compilateur.

La version OPT2 étant celle qui nous a donné jusque là les meilleurs résultats nous allons partir d'elle et essayer d'améliorer le code.

Optimisation 3 OPT3 : OPT2 +alignement des tableaux sur des limites de 32 octets+loop-tiling sur la boucle de sobel_baseline

D'après le rapport maqao de la fonction process_baseline lors de l'optimisation 2, si on aligner nos tableaux sur des limites de 32 bits on pourra gagner en performance pour se faire nous allons en ajouter la 3X la directive suivante sur les 3 tableaux :

__builtin_assume_aligned (&frame[x + 0], 32);

__builtin_assume_aligned (&frame[x + 1], 32);

__builtin_assume_aligned (&frame[x + 2], 32);

Ensuite nous allons allons appliquer le méthode du loop-tiling sur la boucle de la fonction sobel_baseline.

Le loop-tiling, également connu sous le nom de blocage de boucle, est une transformation de boucle qui exploite la localisation spatiale et temporelle des accès aux données dans les nids de boucle. Cette transformation permet d'accéder aux données en blocs (tuiles), la taille de bloc étant définie comme paramètre de cette transformation. Chaque boucle est transformée en deux boucles: une itération à l'intérieur de chaque bloc (intratile) et l'autre itération sur les blocs (intertile).

- **Avant loop-tiling**

```
//
 for (u64 j = 0; j < ((W * 3) - 3); j++)
  for (u64 i = 0; i < (H-3); i++)
    {
  gx = convolve_baseline(&cframe[INDEX(i, j, W * 3)], f1, 3, 3);
  gy = convolve_baseline(&cframe[INDEX(i, j, W * 3)], f2, 3, 3);
```

- **Après loop-tiling**

```
nombre_1=(H - 3);
nombre_2=W * 3) - 3);
assert(nombre_1%TILE==0);
assert(nombre_2%TILE==0);

for (u64 ii = 0; ii < nombre_1; ii+=TILE)
  for (u64 jj = 0; jj < nombre_2; j+=TILE)
    for(u64 i = ii; i < ii+TILE; i++)
      for (u64 j = jj; j < jj+TILE; j++)
        {
          gx = convolve_baseline(&cframe[INDEX(i, j, W * 3)], f1, 3, 3);
          gy = convolve_baseline(&cframe[INDEX(i, j, W * 3)], f2, 3, 3);
```

A l'exécution nous obtenons les résultats suivants :

Flags -OPT2	Average des cycles
O1	7273945,5
O2	6967750
O3	2322089,5
Ofast	2374840

Flags -OPT3	Average des cycles
O1	3421495,5
O2	3696937
O3	1158672
Ofast	1112579

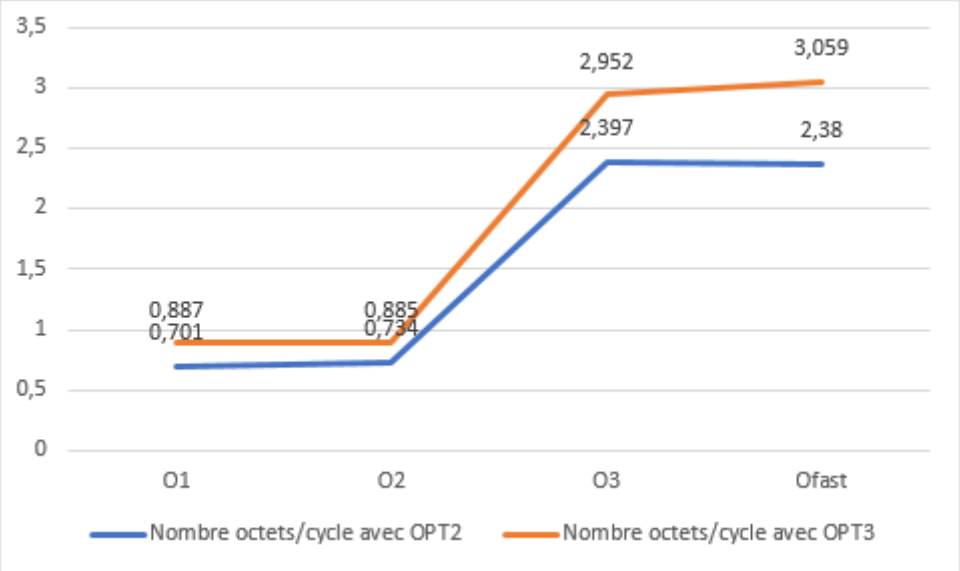

Illustration 8: Courbe de comparaison du nombre de données traitées avec OPT2 et OPT3

L'optimisation 3 nous a permis de traiter plus de données par cycle en même temps on sollicite beaucoup moi les ressource les mesures sur le cpu sont réduits à moitié pratiquement.

Optimisation 4 : Rajout des directives OpenMP

Dans toutes les optimisations que nous avons faites c'est OPT3 qui donnait les meilleurs résultats surtout avec la vectorisation de la boucle process_baseline. Nous allons maintenant appliquer les directives de OpenMP à notre code pour paralléliser certaines régions et observer leur impact sur le code :

Nous allons appliquer ces directives sur notre optimisations OPT3 :

```
#pragma omp parallel for num_threads(2)
for (u64 i = 0; i < fh; i++)
  for (u64 j = 0; j < fw; j++)
   #pragma omp critical
    r += m[INDEX(i, j, fw)] * f[INDEX(i, j, fw)];
```

Ajout des directives OpenMP à la boucle de la fonction convolve_baseline dans sobel.c

```
assert(nombre_2%TILE==0);
#pragma omp parallel for num_threads(2)
for (u64 ii = 0; ii < nombre_1; ii+=TILE)
  for (u64 jj = 0; jj < nombre_2; j+=TILE)
    for(u64 i = ii; i < ii+TILE; i++)
      for (u64 j = jj; j < jj+TILE; j++)
        #pragma omp critical
```

Ajout des directives de OpenMP à la boucle de la fonction sobel_baseline dans sobel.c

Nous obtenons les résultats suivants :

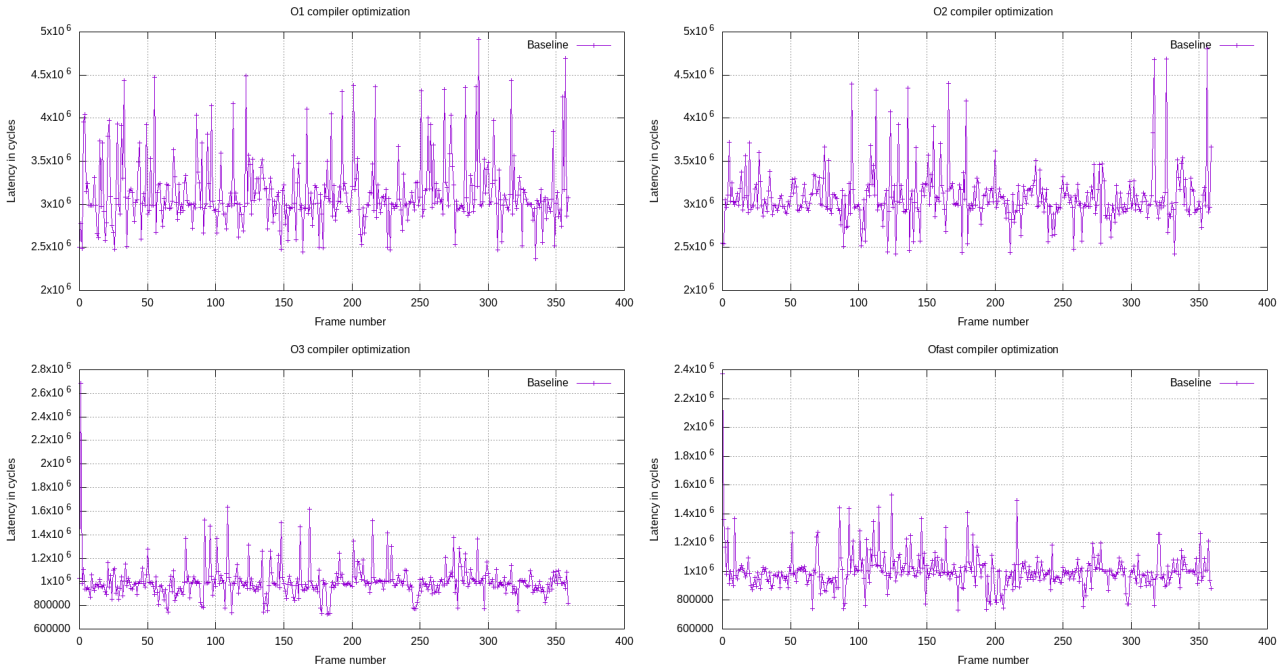

Illustration 9: Les Courbes de la latence dans le cycle en fonction du nombre de frames de la version OPT4 selon les différents flags

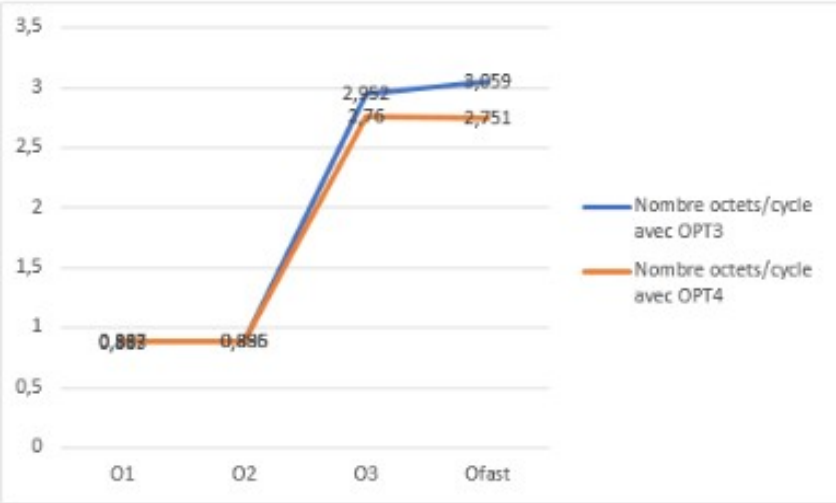

Illustration 10: Courbes comparatives des metriques obtenues entre OPT3 et OPT4 sous gcc

Flags OPT3	Average des cycles
O1	3421495,5
O2	3696937
O3	1158672
Ofast	1112579

Flags OPT4	Average des cycles
O1	3639084,5
O2	3612695
O3	1704871
Ofast	1551704

Nous avons remarque que lorsqu'on a ajouté les directives sur la fonction nous avons perdu en performance, nous avons traité moins de données par cycle.
La parallélisation nous a fait perdre en performance.

Dans nos analyses avec maqao, celui-ci nous avait suggéré d'utiliser un autre compilateur afin de pouvoir gagner en performance.
Ainsi Nous allons maintenant refaire nos mêmes analyses et optimisations avec le compilateur d'intel icc.

Autres compilateurs et flags de compilation

Nous allons reboucler à la première étape c'est à dire exécuter le code de base mais cette fois-ci avec le compilateur icc. Dans notre makefile CC=gcc devient CC=icc on enlève l'option -g3

Analyse et Mesures des Performances du code original avec ICC

Environnement & Portabilité de icc

Notre environnement de travail reste le même, c'est à dire sur un processeur Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz, Le système d'exploitation est un Ubuntu en 64 bits installé en Dual Boot (linux natif) ce qui facilite grandement les mesures de performances et permet l'utilisation de Likwid.

Portabilité de icc

Le compilateur Intel ® C ++ peut générer du code pour les applications d'architecture IA-32, Intel64 ou IA-64 sur tout système Linux * basé sur Intel ®. Les applications d'architecture IA-32 (32 bits) peuvent fonctionner sur tous les processeurs Intel Systèmes Linux. Les applications d'architecture Intel64 et les applications d'architecture IA-64 ne peuvent s'exécuter que sur les systèmes Linux à architecture Intel64.

La version de notre compilateur étant icc 19.1.1.217 compatible sur l'architecture ADM64 (aka x86_64) dont nous disposons.

Analyse des performances du code original avec icc et comparaison avec résultats sous gcc

Nous allons maintenant recompiler les optimisations que nous avons faites avec le compilateur icc.

On s'assure que notre ordinateur est sous secteur, et que la fréquence du processeur est fixe au maximum durant les mesures en le spécifiant manuellement avec la commande : **cpupower -c 0-3 frequency-set -u 3.3GHz**

A l'exécution nous obtenons les résultats suivants :

Flags / icc	Average des cycles	Nombre octets par cycle	Déviation standard
-O1	4273168	0,796	13,278
-O2	6216602	0,516	7,592
-O3	2994957,5	2,754	25,380
-Ofast	1403889	2,725	15,996

Tableau des métriques obtenus avec le compilateur icc sur le code sans optimisation

Flags / gcc	Average des cycles	Nombre octets par cycle	Déviation standard
-O1	4759504	0,751	14,808 %
-O2	5678126	0,789	14,645 %
-O3	4239194	0,800	11,553 %
-Ofast	4604360,5	0,789	13,754 %

Tableau des métriques obtenus avec le compilateur gcc sur le code sans optimisation

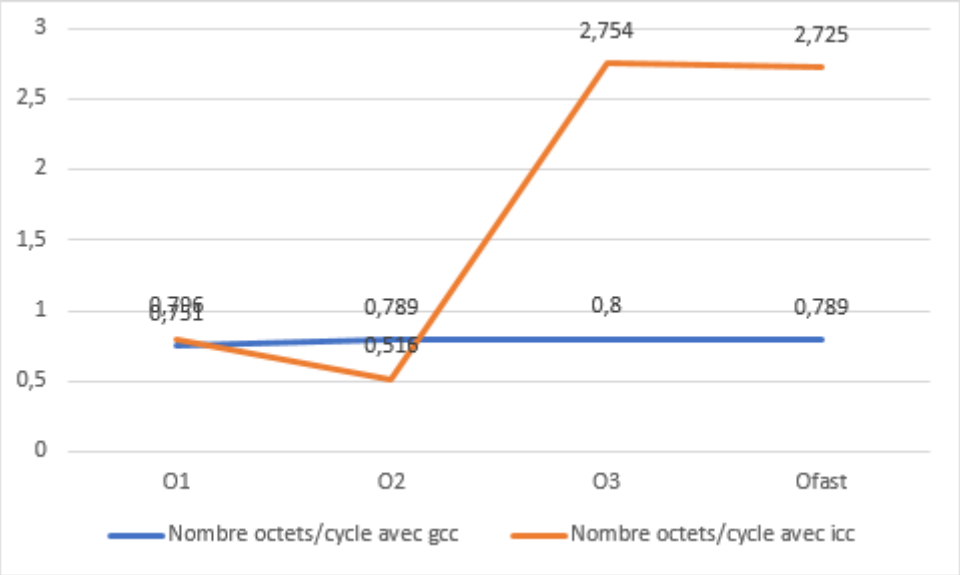

Illustration 10: Courbe comparaison du nombre de données traitées par cycle sur le code original entre icc et gcc

interprétations des résultats

Nous observons qu'avec le compilateur icc nous avons traités plus 3 fois plus de données par cycle pour les flags O3 et Ofast et on sollicite 2X moins les ressources. Cependant pour les flags O1 et O2 les résultats donnaient avec une légère différence les mêmes résultats.

Rapport maqao

Nous allons a présent générer le rapport maqao.

Le rapport d'optimisation maqao indique au programmeur quelles optimisations ont été effectuées et précise également pour quelle raison d'autres ne l'ont pas été. Ces informations peuvent être utilisées pour ajuster le code et améliorer d'avantage la performance de l'application.

Le rapport maqao nous a révélé les mêmes optimisations et solutions que pour l'analyse sous gcc. Ainsi de là je peux dire que je vais refaire les mêmes optimisations que pour précédemment.

La section ci-après va regrouper l'ensemble des comparaisons des résultats des différentes optimisations effectuées entre gcc et icc.

Comparaison des optimisations sous gcc et sous icc

- **OPT1 gcc vs OPT1 icc**

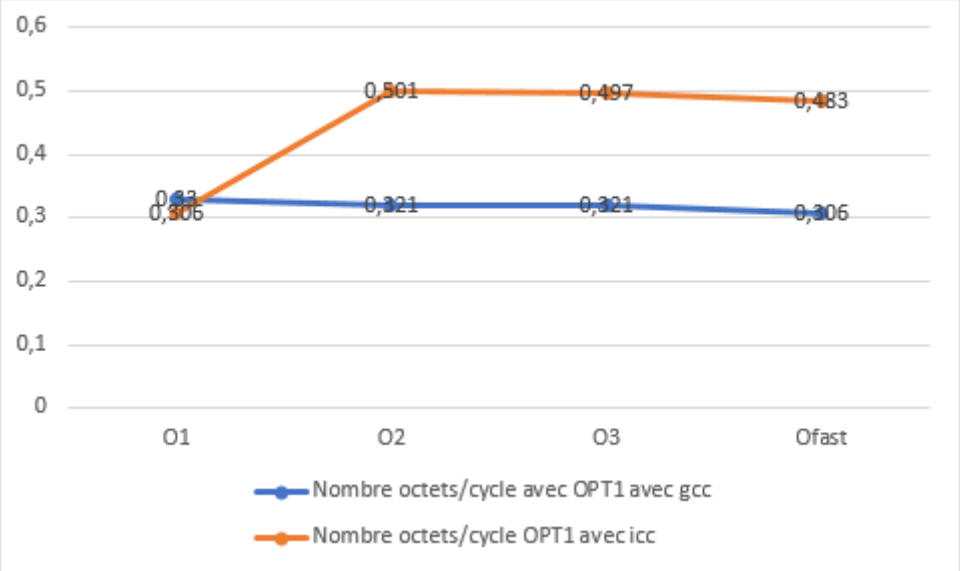

Illustration 11: Courbes du nombre de données traités par cycle avec OPT1 entre gcc et icc

- **OPT2 gcc vs OPT2 icc**

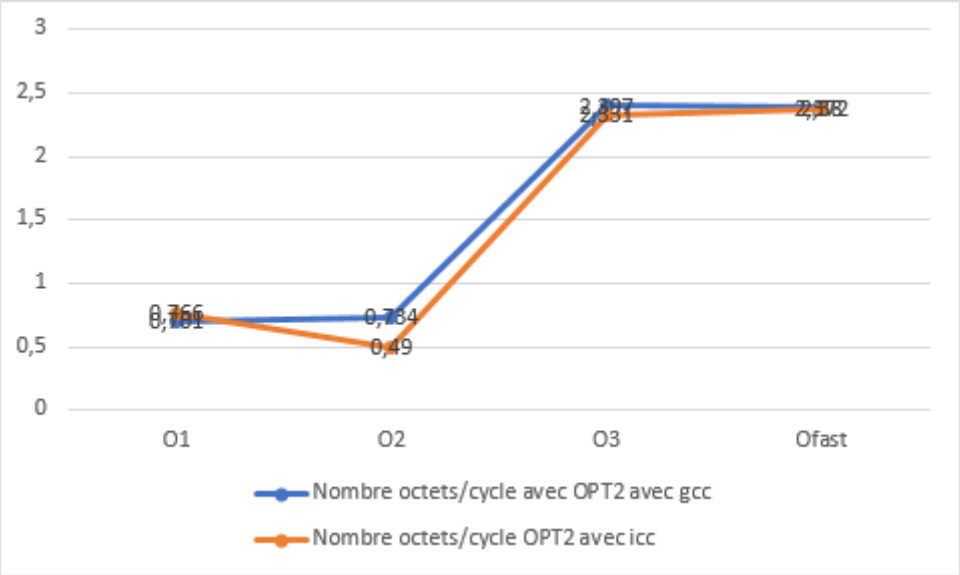

Illustration 12: Courbes du nombre de données traités par cycle avec OPT2 entre gcc et icc

- **OPT3 gcc vs OPT3 icc + ajout des directives OpenMP (OPT4)**

Lors de la compilation, l'alignement des tableaux sur 32bits ne fonctionnait pas sur on a du l'enlever dans la fonction process_baseline pour que cela puisse compiler.

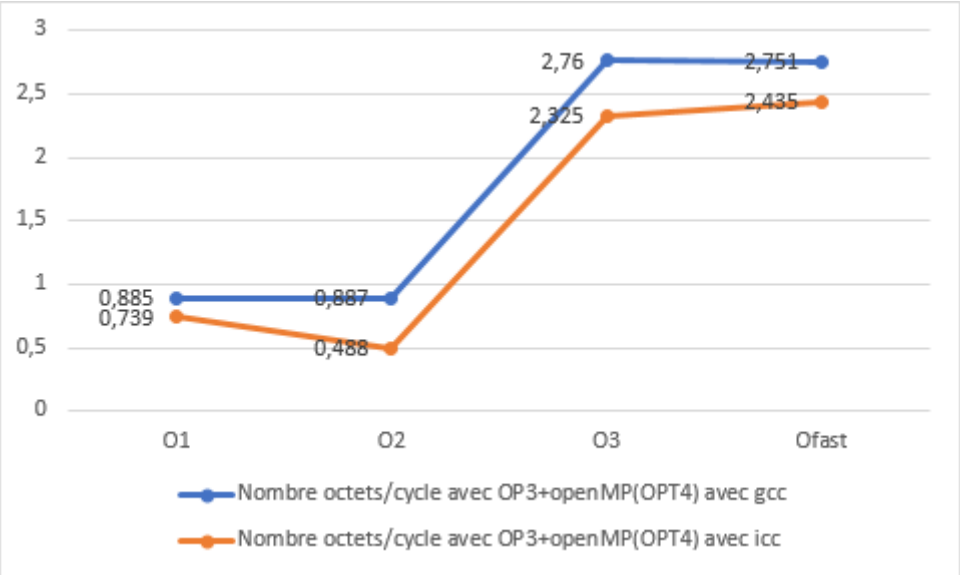

Illustration 13: Courbes du nombre de données traités par cycle avec OPT3+ajout de OpenMP(OPT4) entre gcc et icc

Interprétations des résultats :

Nous avons constaté que sur le compilateur icc certains flags de compilation gagné en performance et cela nous permettait d'avoir un code beaucoup plus performants mais dans certaines mesures c'était gcc qui donnait les meilleurs résultats mais néanmoins icc reste le meilleur d'après nos analyses en termes de données traités et d'utilisation des ressources.
L'ajout des directives openMp nous faisait perdre en performance.
Le compilateur icc était plus favorable aux flags O3 et Ofast

Autres flags de compilation

Jusque là nous avons fait nos tests avec les flags suivantes : O1, O2, O3, Ofast nous allons maintenant tester d'autres flags de compilations avec gcc.

L'optimisation OPT3 était celle qui nous avait donné les meilleurs résultats sur notre code, nous allons changer nos flags de compilations et voir si le code sera mieux optimisé ou pas.

Nous résumons les résultats obtenus dans le tableau suivant :

Options compilations	flags	Nombre d'octets traités par cycle	Average des cycles
-xHost : Peut générer des instructions pour le jeu d'instructions et le processeur les plus élevés disponibles sur l'hôte de compilation	O1 O2 O3 Ofast	0.778 0.487 3.235 3.208	14787084 6517096 1116809 1096172
-no-prec-div : accélér les divisions par remplacement avec multiplications par l'inverse ou utilisation d'instructions moins précises mais plus rapides	O1 O2 O2 Ofast	0.799 0.519 3.071 2.545	4345362 6171764 1139063 2704191
-qopt-report=2 -vec-threshold0 -O3	O1 O2 O3 Ofast	0.780 0.495 2.480 2.399	4609030 9972193 2162446 2378324
-O3 -fp-model fast=1 -xHost :contrôle la sémantique des calculs à virgule flottante.	O1 O2 O3 Ofast	0.724 0.465 2.514 2.587	5580234 9974127.5 1510946 1205836

Permet des optimisations plus agressives sur les données à virgule flottante			

Tableau comparatif des métriques obtenues avec le compilateur icc avec différents flags d'optimisation

Parmis les flags que nous avons utiliser, xHost nous a donné les meilleurs résultats même si la différence n'est pas très significative avec les résultats obtenus avec OPT3 sur icc.

Synthèse

Durant cet étude nous avons été amené à analyser et optimiser les performances des codes qu'on nous avait fournis. Nous avons vu que à l'aide des outils comme maqao on pouvait avoir un code beaucoup plus performants en lisant les rapports sortis à l'exécution.

Dans notre cas c'est surtout lorsque nous avons vectoriser notre boucle process_baseline + la méthode du loop tiling sur sobel_baseline que nous avons eu les meilleurs performances sous gcc.

Mais avec le changement de compilateurs comme icc, on a vu que sur les méthodes d'optimisations que nous avions utiliser on a traité plus de données par cycle et en même temps on sollicitait moins les ressources de l'ordinateur. Avec le compilateur icc les flags O3 et Ofast bénéficiaient plus des optimisations.

Nous avons aussi constater que l'ajout des directives OpenMP n'a pas permis de gagner en performance au contraire ça nous a fait perdre performance.

Conclusion

Durant ce projet nous avons essayé d' analyser et d'optimiser des codes en réalisant plusieurs tests sur différentes versions que nous avons implémenter.Il nous a permis de mieux appréhender les outils pour faire une analyse et optimisation de code.

Il nous a aidé à mettre en pratique ce qu'on a acquis pendant les cours et Tds du cours d'AOB.

Références et bibliographie :

https://fr.wikipedia.org/wiki/Optimisation_de_boucle

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

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

http://forums.cirad.fr/logiciel-r/viewtopic.php?t=6097

Cours AOA