Rapport Architecture parallele

FILTRE DE SOBEL

- Thomas ROGLIN

Introduction : Filtre de Sobel

Le filtre de sobel est un coeur de calcul utilisé pour la détection de contour. Ce filtre, appliqué à une image, calcul le gradient de chaque pixel, en comparant leur intensité. Il permet de faire ressorir les zones avec de fortes différences d'intensités, autrement dit, les contours .

1) Objectif et méthodologie ?

L'objectif principal que nous nous fixons dans ce rapport est d'optimiser la fonction sobel_baseline. Pour cela nous utiliserons divers outils tel que la parallelisation et la vectorisations à l'aide des fonctions intrinsiques d'intel

2) Compilateurs et environement

2.1) Compilers

- gcc (GCC) 12.2.1
- clang version 15.0.7
- AMD clang version 13.0.0 (CLANG: AOCC_3.2.0-Build#128 2021_11_12) (based on LLVM Mirror.Version.13.0.0)

2.2) Environment

- La fréquence du cpu a été réglée a son maximum grace au gouverneur 'performance' de cpupower
- Le Turbo boost est activé

2.3) Architecture

- CPU name : AMD Ryzen 5 PRO 4650U
- MicroArchitecture : Zen2
- Cores per socket : 6
- Threads per core : 2
- Cache line size : 64 Bytes
- Level 1 cache size: 32*6 KiB
- Level 2 cache size: 512*6 KiB
- Level 3 cache size: 4MiB * 2 (shared between 3 cores)

- Level 3 nb ways : 16
- RAM : 16 GiB
- Instruction sets : SSE, AVX2 (16 registres YMM et XMM)
- Min Frequency: 1.4 GHZ
- Max Frequency: 2.1 GHZ
- Turbo Frequency: 4.0 GHZ

3) Présentation de la baseline

Le code appliquant le filtre de sobel fournis dans la baseline que nous devons optimiser contient 2 fonctions. Une premiere, `sobel_baseline` parcourant tout les pixels de la frame. Pour chaque pixels, elle appelle la fonction `convole_baseline` deux fois, calcule la norme et la stocke dans une deuxieme frame. La fonction `convolve_baseline` prend en parametre l'adresse d'un pixel, un filtre et la taille du filtre. Au moyen de 2 boucles imbriquées, elle applique le filtre sur les voisins du pixels, puis renvoie le resultat.

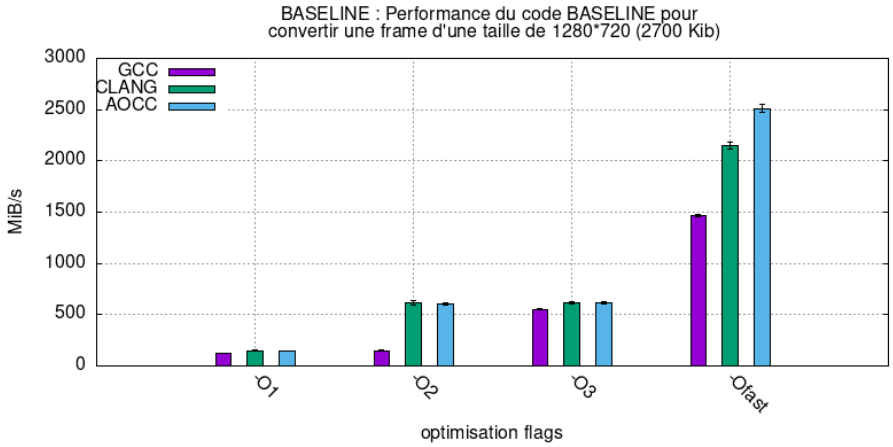

Figure 1: benchmark de la baseline

Table 1: BandeWidth de la baseline en Mib/s

FLAGS	GCC	CLANG	AOCC
O1	121	148	148
O2	148	612	607
O3	550	615	614
Ofast	1468	2151	2511

Le code de la baseline est fournis avec un makefile, dont les options de compilations par défault sont : `-march=native -O1`, avec le compilateur `gcc`. Nous

prendrons donc les performances issue de l'execution de la baseline compilée avec ces informations, comme bande passante de référence, c'est à dire $121Mib/s$.

Dans les 2 figures ci dessus, nous observons la bande passante en Mib/s. Nous y comparons l'effet des flags `Ox` et des compilateurs sur la baseline (l'option `-march=native` est toujours activée).

On que le flag `Ofast` permet d'obtenir une bande passante allant de $1500Mib/s$ à $2500Mib/s$. En `Ofast`, les compilateurs `gcc`, `clang` et `aocc` permettent respectivement une accélération d'environ 12, 18 et 21.

4) Première Optimisations

4.1) SQRT

La racine carrée est une opération couteuse pour un programme. D'après la table d'instruction présenté par agner Fog, l'instruction $fsqrt$ a coute 22 cycles cpu à s'executer sur une architecture Zen2. Cela fait de cette instruction un potentiel bottleneck pour le programme.

Pour optimiser cette section du programme, nous présenterons 2 solutions après avoir rappelé le contexte de cette racine carrée. Tout d'abord, une magnétude mag est calculé grace à cette racine carrée `mag = \sqrt{ {g_x}^{2} + {g_y}^2 }`. Et si cette magnétude est supérieur à un certain threashold, alors ce pixel prend la valeur 255, sinon il prend la valeur mag:

$$pixel = (mag > threshold)\ ?\ 255 : mag;$$

La premiere solution proposée est de supprimer les nuances de gris sur l'image finale. Ainsi, nous pouvons calculer `mag2 = {g_x}^{2} + {g_y}^2`, puis

$$pixel = (mag2 > threshold^2)\ ?\ 255 : 0;$$

Nous nous somme débarassé de la racine carrée, en échange de la perte des nuances de gris sur l'image. Cependant cette perte sera considérée comme négligeable étant donnée que le coeur de l'algorithme, la détection de contours, reste présent. (le calcul $mag2 > threashold^2$ n'est pas exacte et est une approximation.)

La 2^{ème} solution que nous proposons est d'utiliser une approximation de la racine carrée. une instruction telle que `vrsqrtps` propose cela, pour une précision de 10^{-12}, et pour un cout de 3 à 4 cycles. Cela nous permet donc de concerver des nuances de gris à un cout 8 fois plus faible. Il est possible de l'utiliser en demandant au compilateur d'appoximer les racines carrées avec `-mlow-precision-recip-sqrt` , ou, avec intrinsique, en utilisant `_mm256_rsqrt_ps`.

Pour la suite du rapport, et pour les figures que nous présenterons ce dessous, nous choisirons la premiere option consistant à supprimer la racine carrée et les nuances de gris.

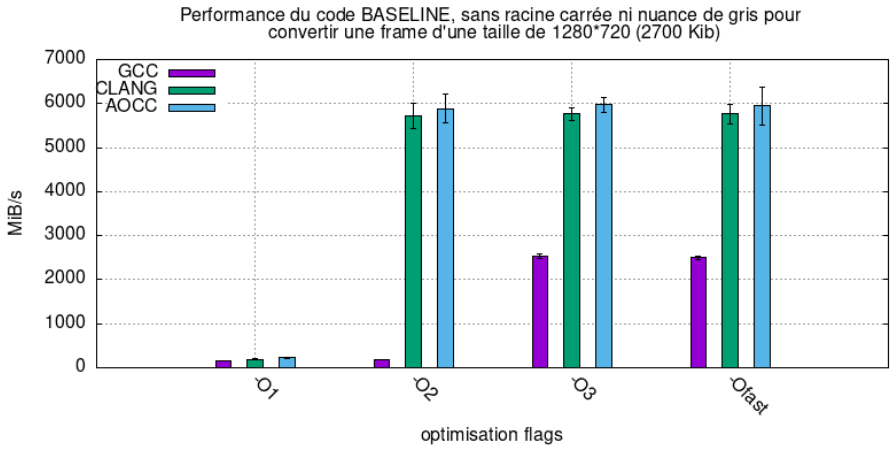

Figure 2: benchmark de la baseline sans racine carrée

Table 2: BandeWidth de la baseline sans sqrt ni nuance de gris en Mib/s

FLAGS	GCC	CLANG	AOCC
O1	153	192	222
O2	190	5716	5886
O3	2530	5761	5972
Ofast	2498	5759	5946

Nous observons globalement une grosse amélioration dans la bande passante. `Clang` et `AOCC` offrent une performances relativement identiques, grosse amélioration $*30$ de `O1` à `O2`, puis une bande passante d'environ `5800` pour `O2 O3 Ofast`. La déviation standard étant élevée, il n'y a pas de différences assez significatives pour les départager. `gcc` en revanche obtient une performance de 2500 Mib/s$, ce que est 2 fois plus faible que les autres compilateur. Cette optimisation est efficace, elle permet d'obtenir avec `Ofast` une accelération respective pour `gcc` `clang` et `aocc` de 21 47 et 46.

4.2) Diminution la taille de la frame

La matrice fournit dans le code de la baseline présente une redondance des données. En effet, le filtre gris remplace les composantes rgb par des composantes par 3 composantes grises ggg ayant la meme valeur. En supprimant les deux

composantes inutiles, on divise tout d'abord la taille de la matrice par 3, et on permet aussi à chaque cache line chargée depuis la DRAM de ramener 3 fois plus d'éléments utile au calcul. Cela permettra au cpu et à nous aussi de vectoriser plus facilement le programme.

Afin d'implémenter cela, nous allons modifier la fonction grayscale qu'elle écrive dans la matrice seulement une composante. Aussi, nous ajouterons une fonction avant l'écriture dans le fichier pour réagrandir l'image pour que l'écriture se passe correctement. (cette fonction ne comptera pas dans notre mesure du temps)

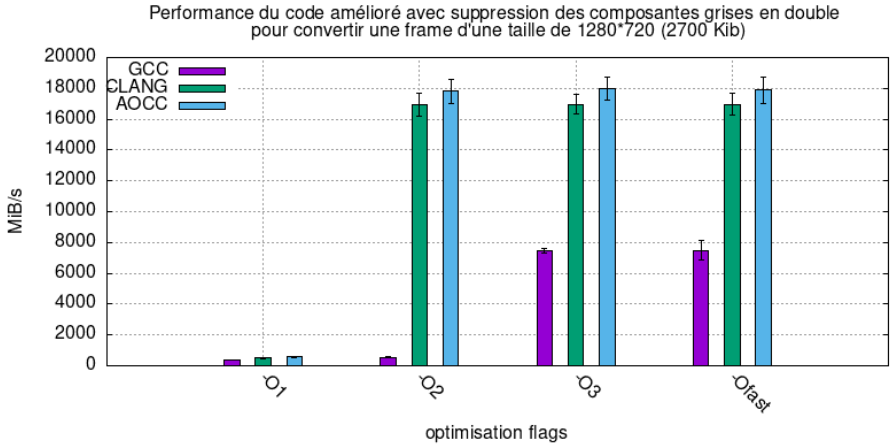

Figure 3: benchmark sans les composantes grises en double, avec des éléments de 8 bits

Les résultats de cette optimisation sont présentés dans le graphique ci dessus. Ici la frame est composée d'éléments de 8 bits.

Dans la figure 4 sont présentés le meme benchmark sauf qu'ici chaque frame est composée d'éléments de 32 bits.

NOTE :

Lorsque le code a été testé pour la premiere fois, aucune différence significtive n'est de performance ressortie. Ceci est du à une erreur de ma part.

C'est pourquoi pour la suite du rapport nous travaillerons sur des données de 32 bits et non des données de 8 bits comme le graphe précedent le suggerait.

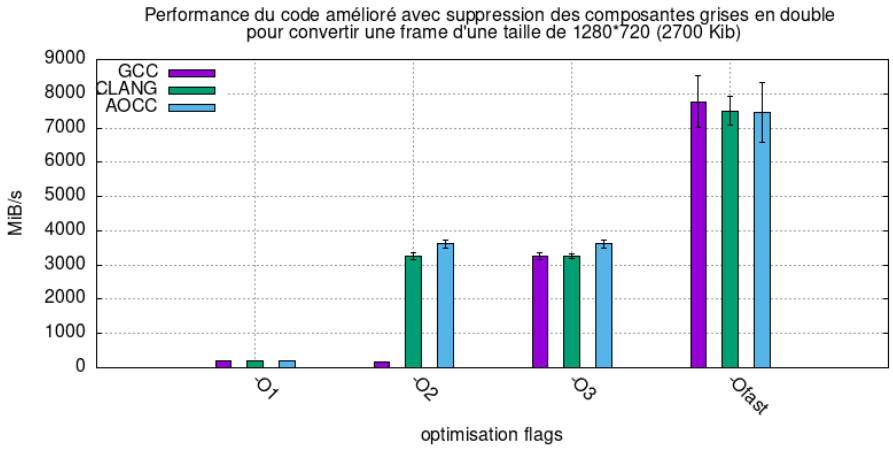

Figure 4: benchmark sans les composantes grises en double, avec des éléments de 32 bits

Table 3: BandeWidth du filtre de sobel, optimisé sans racines carrées, sans doublon dans les données, avec des éléments de 32 bits, en Mib/s

FLAGS	GCC	CLANG	AOCC
O1	185	190	197
O2	183	3251	3618
O3	3263	3253	5972
Ofast	7760	7504	7454

4.3) Unroll et inline de convolve baseline

Nous travaillons uniquement uniquement avec un filtre de sobel $3 * 3$, le nombre d'iteration de la fonction `convole_baseline` est déja connu à l'avance. Dérouler entierement la fonction nous évite 9 jumps, une instruction causant que le cpu gère mal.

Aussi, comme le filtre est déja connu, on peut s'adapter à lui directement dans le code. En remplacant `filtre[x]` directement par sa valeur, on diminue les accès mémoire. On peut aussi supprimer directement les étapes ou la valeur du filtre est à zero.

De plus, la taille de l'image étant connu par avance, on peut précalculer le délage d'adresse nécessaire pour acceder aux pixels voisins.

En sommants toute ces optimimisations, on obtient donc le code suivant :

```
for (u64 i = 0; i < (H - 3); i++)
```

```
    for (u64 j = 0; j < (W  - 3); j++) {

        gx = 0;
        gy = 0;

        u64 idx = INDEX(i, j, W );

        gx += frame[idx + IDX_1] * (-1);
        gx += frame[idx + IDX_3] * ( 1);
        gx += frame[idx + IDX_4] * (-2);
        gx += frame[idx + IDX_6] * ( 2);
        gx += frame[idx + IDX_7] * (-1);
        gx += frame[idx + IDX_9] * ( 1);

        gy += frame[idx + IDX_1] * (-1);
        gy += frame[idx + IDX_2] * (-2);
        gy += frame[idx + IDX_3] * (-1);
        gy += frame[idx + IDX_7] * ( 1);
        gy += frame[idx + IDX_8] * ( 2);
        gy += frame[idx + IDX_9] * ( 1);


        mag = gx * gx + gy * gy;

        Bframe[  INDEX(i+1,j+1,W)  ] = (mag > 10000) ? 255 : 0;
}
```

L'écriture `a += a * c` permettra normalement au compilateur de remplacer la multiplication et l'addition par une seule instruction `fma` moins couteuse que les deux cumulés.

ces optimisations sont implémentés dans la fonctions `sobel_opti_v1`. Ci dessous sont présenté le benchmark de cette fonction.

Table 4: BandeWidth du code inliné, unrollé et précalculé en Mib/s

FLAGS	GCC	CLANG	AOCC
O1	2255	2451	1877
O2	2623	8605	8631
O3	8835	8753	8502
Ofast	8821	8828	8968

Les résultats que nous obtenons ici montre une amélioration par rapport à l'optimisation précedente.

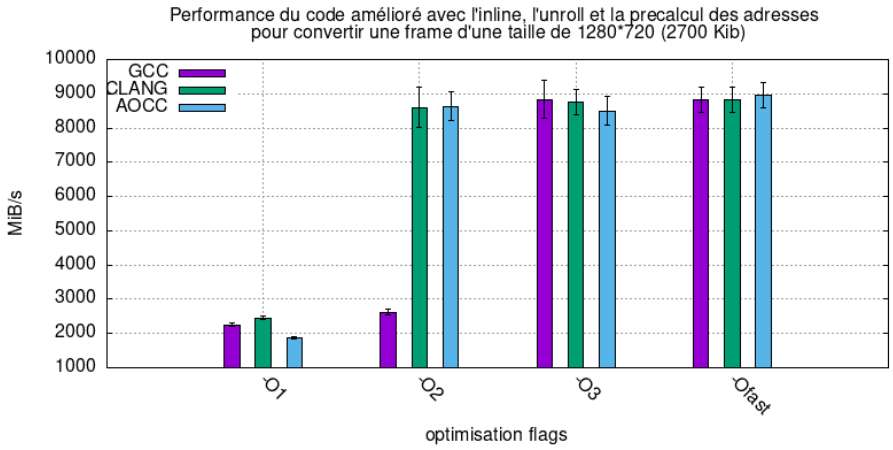

Figure 5: benchmark du code inliné, unrollé et précalculé

Finalement, nous avons pu atteindre une accélération de 73 avec `GCC` et `clang`, et de 74 avec `AOCC`.

5) SIMD

La vectorisation est normalement gérée automatiquement pas le compilateur à partir de `-O2`. Cependant nous allons tout de meme essayer d'améliorer ce que peut faire le compilateur en utilisant les intrinsiques fournie par intel.

Zen2 n'étant pas compatible avx512, nous n'utiliserons que AVX2 et des registres 256 bits.

La question de la représentation des données en mémoire s'est posée. AVX2 ne permet pas de manipuler des données 8 bits. Utiliser des données de 16 bits est risqué, car, d'après le filtre f1, la valeur maximale que peut prendre gx est $255 * 4$, et mettre cette valeur au carrée provoquerait un overflow (pas tout à fait un problème en réalité, ce point sera abordé dans la partie perspective d'amélioration de ce rapport).

Nous choisissons donc d'utiliser des données d'une taille de 32 bits, pour cela nous devons changer le type des matrices que l'ont traite pour correspondre.

La stratégie adoptée pour vectoriser le code est Pour vectoriser cet algorithme, nous avons choisi la stratégie suivante: A l'étape i, j, avec i la i_{eme} ligne et j la j_{eme} colonne, nous chargons dans un registre ymm les 8 éléments à partir de l'adresse (i, j), et nous conciderons que ce vecteur contient les éléments correspondant au premier element du filtre. Nous repetons cette étape en chargeant les éléments à partir de l'adresse $(i, j + 1)$, et on concidere que ce vecteur contient les éléments correspondant au deuxieme element du filtre. On

continue cette étape pour les neufs éléments allant de (i, j) à $(i + 2, j + 2)$. Il nous suffit ensuite d'accumuler dans 2 vecteurs g chaque vecteur multiplié par son élément du filtre correspondant.

à l'itération i, j on a donc :

“‘c

```
u64 idx = INDEX(i, j, W );

__m256 _a1 = _mm256_loadu_ps( &Aframe[ idx + IDX_1 ] );
__m256 _a2 = _mm256_loadu_ps( &Aframe[ idx + IDX_2 ] );
...
__m256 _a11 = _mm256_loadu_ps( &Aframe[ idx + IDX_8 ] );
__m256 _a12 = _mm256_loadu_ps( &Aframe[ idx + IDX_9 ] );


__m256 _gx = _mm256_setzero_ps();
__m256 _gy = _mm256_setzero_ps();

_gx =
    _mm256_add_ps( _mm256_mul_ps (_a1, -1 ),
    _mm256_add_ps( _mm256_mul_ps (_a3,  1 ),
    _mm256_add_ps( _mm256_mul_ps (_a4, -2 ),
    _mm256_add_ps( _mm256_mul_ps (_a6,  2 ),
    _mm256_add_ps( _mm256_mul_ps (_a7, -1 ),
    _mm256_add_ps( _mm256_mul_ps (_a9,  1 ), _gx))))));

_gy =
    _mm256_add_ps( _mm256_mul_ps (_a1,  1 ),
    _mm256_add_ps( _mm256_mul_ps (_a2, -2 ),
    _mm256_add_ps( _mm256_mul_ps (_a3, -1 ),
    _mm256_add_ps( _mm256_mul_ps (_a7,  1 ),
    _mm256_add_ps( _mm256_mul_ps (_a8,  2 ),
    _mm256_add_ps( _mm256_mul_ps (_a9,  1 ), _gy))))));

_gx = _mm256_mul_ps( _gx, _gx );
_gy = _mm256_mul_ps( _gy, _gy );
_gy = _mm256_add_ps( _gy, _gx );


_gx= _mm256_cmp_ps( _threshold2, _gx, _MM_CMPINT_LE);
_gx = _mm256_add_ps( _b1, _255 );
_mm256_store_ps( &Bframe[ idx ], _b1 );
```

‘

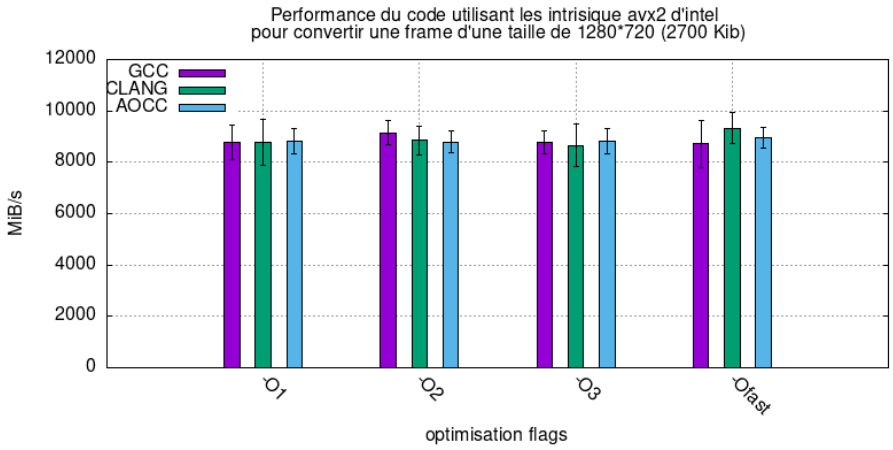

Figure 6: benchmark de la version avx2 du code

Table 5: Bandewidth en Mib/s du code vectorisé avec les intrinsiques avx2

FLAGS	GCC	CLANG	AOCC
O1	8790	8772	8825
O2	9153	8846	8789
O3	8773	8658	8822
Ofast	8720	9332	8935

On peut donc comparer la version vectorisée manuelement, avec le notre précédent code optimisé vectorisé par le compilateur.

Table 6: Comparaison de l'accélération entre les versions intrisique et non intrinsique (table 4 et 5)

FLAGS	GCC	CLANG	AOCC
O1	x3.9	x3.5	x4.7
O2	x3.4	x0.97	x1.01
O3	x0.99	x0.99	x1.03
Ofast	x1.01	x0.94	x0.99

En regardant le tableau d'accélération, on remarque que la vectorisation manuele à l'aide des intrinsiques n'est pas plus efficace que la version vectorisé par le compilateur. Il est difficile de faire une comparaison très précise etant donnée la très forte déviation standard de la version intrinsique.

Aussi, on peut quand meme remarquer que notre code intrinsique fonctionne environ 4 fois mieux en `-O1`. Cela semble logique, en effet la vectorisation est activée à partir de `-O2`. Quant au cas de `gcc` en `-O2`, il semble qu'il n'arrive pas à vectoriser correctement contrairement à `clang` et `aocc`.

6) Optimisation de l'algorithme

6.1) Traitement de plusieurs lignes

Après avoir étudié quelques optimisations du code, nous allons tenter de modfier l'algorithme afin d'essayer d'encore des performances

Tout d'abord, il est bon de remarquer que jusqu'à présent, nous avons travaillé ligne par ligne. Aussi, lorsque nous travaillons sur la ligne i, nous chargeons des éléments de la ligne $i+1$. Ces éléments sont exploités qu'une fois, avant d'etre déchargé puis rechargé à la ligne suivante.

Pour optimiser cela, on va traiter plusieurs ligne à la fois, ainsi, en traitant la ligne i, $i+1$, $i+2$, l'élément à la ligne $i+1$ sera chargé une seule fois puis utilisé pour le calcul de la ligne i, $i+1$ et $i+2$. Toutefois, nous ne pouvons pas traiter autant de ligne que nous souhaitons, car nous somme limité par le nombre de registre YMM, qui sont au nombre de 16. Ci dessous sont présenté les résultats de la version amélioré d'avx traitant 2 lignes à la fois.

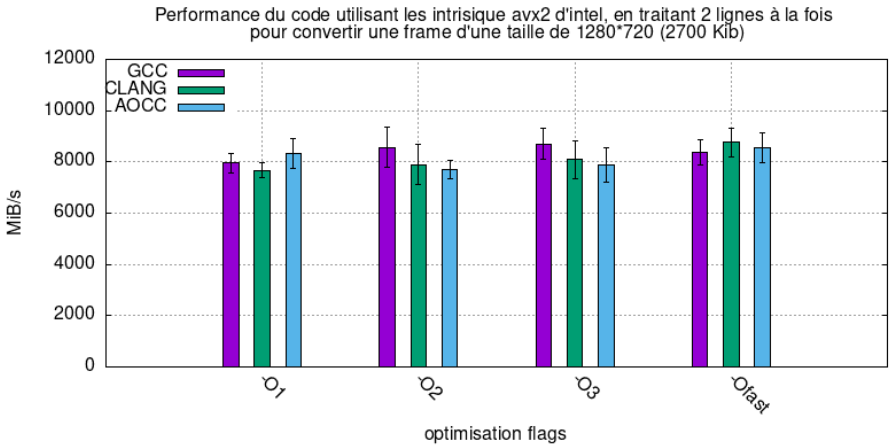

Figure 7: benchmark de la version avx2 traitant 2 lignes à la fois

Les résultats présentés sur le graphique ci dessus ne semblent pas concluant. Nous obtenons environs les memes résultats que sur la version sur une ligne, et la forte déviation standard nous empeche encore de savoir si il y a une amélioration ou non.

6.2) Cache Blocking

Une autre idée d'algorithme est le cache blocking, en travaillant sur une faible section de l'image, on peut s'assurer de ne jamais faire déborder le cache L1. l'implémentation de cette méthode est faite dans la `fontion sobel_CL`.

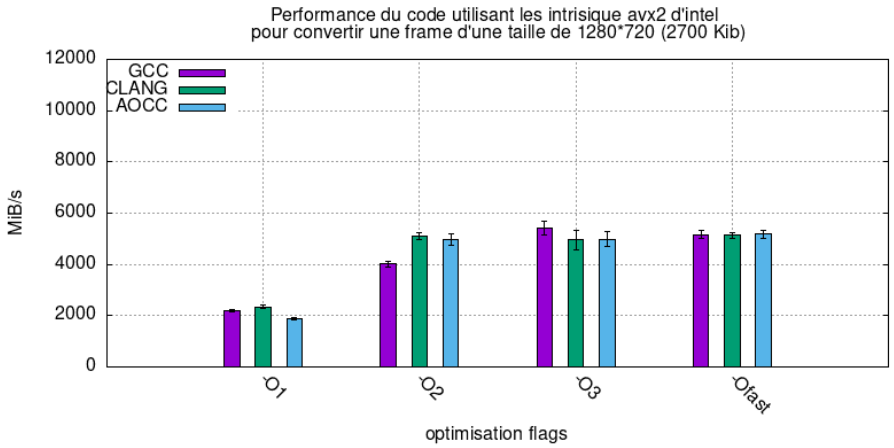

Figure 8: benchmark de la fonction de cache blocking

Les résulats présenté ici sont très peu élevé, et la méthode ne semble pas fonctionner. On peut l'expliquer facilement : Lorsqu'on traite l'image ligne pas ligne, les données de la ligne précédentes sont encore dans le cache L1 car une ligne de l'image est très loin de remplir le cache L1. Ainsi on ne déborde jamais du cache L1, donc le cache blocking perd de son interet.

6.3) kernel de dimension 1

Une autre solution proposée dans un article est de ne plus utiliser de filtre 2D mais d'utiliser deux filtres 1D.

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

En utilisant cette formule, on peut appliquer dans un premier parcours le premier filtre $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$ puis dans un deuxieme parcours le second filtre $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$. Cette fonction est implémenté dans `sobel_1D`.

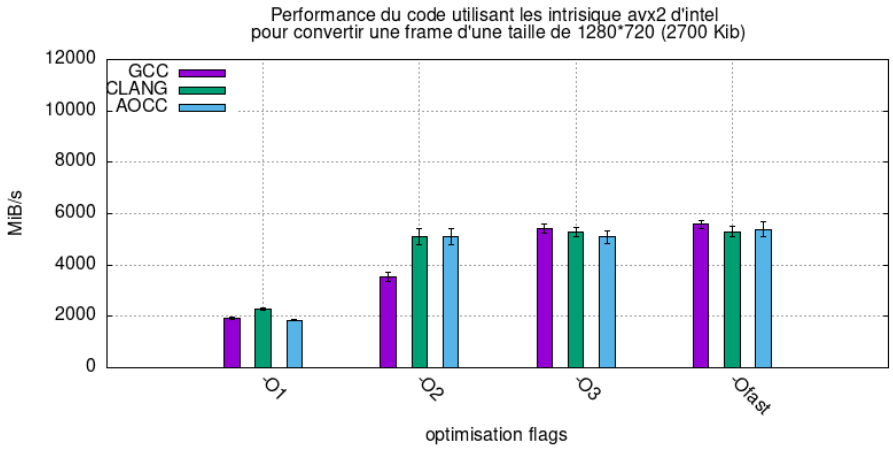

Figure 9: benchmark de la fonction de kernel 1D

Parallelisation

La parallelisation est Une optimisation qui n'a pas pu etre implémenté par manque de temps. La parallelision se prete bien l'application d'un filtre de sobel sur une vidéo. En effet, chaque frame est indépendante de la précedent. Ainsi, pour paralleliser, nous pourrions récupérer toute les frames du fichier raw en le stockant en mémoire grace à nmap. Avec un partage des taches efficace, chaque thread aurait a traiter un meme nombre de frame, puis pourraient les stocker dans un buffer de sortie à l'emplacement du numero de la frame. Une fois toute les frames de la vidéo traités, le buffer de sortie serait écrit dans le fichier out.raw.

Sur cette machine possedant 12 threads, on peut d'attendre à obtenir un boost théorique de *12. La version intrinsiques ayant une accélération de 70 par rapport à la baseline, on peut s'attendre à une accéléartion pouvant aller jusqu'à x840 avec une version intrinsique parallelisée.

Autres optimisation

D'autres optimisations n'ayant pas été explorée sont possible. Dans ce rapport, nous nous sommes concentré sur l'optimisation de la fonction de sobel. Cependant, les autres éléments du code pourraient etre amélioré. Tout d'abord les IO, Il pourrait peut etre etre plus efficace de charger en mémoire toute la vidéo d'un coup à l'aide de nmap. Aussi, la fonction grayscale peut etre optimisée, tout comme la fonction reagrandissant l'image .

Aussi, choisir de représenter la frame pas une matrice de `f32` au lieu de rester avec `u8` a eu un fort cout en performance. Rester avec des `u8` aurait été préférable,

bien que le code intrinsique aurait été plus compliqué.

Conclusion

A travers diverse optimisation telle que la suppression de la racine carrée, la suppression des composantes dupliquées, l'unroll et le précalcule des index de convolve_baseline, nous avons atteins une accélération maximale de x74. Cette derniere a été atteinte avec `AOCC` et `-Oflags`. Cela représente une bande passante de 9Gib/s. La taille d'une frame étant de $1080 * 720 * 3 bytes$, c'est à dire 2.6Mib, le filtre est donc capable de traiter 3400 images par secondes.