Rapport Architechture Parallele

Rodolphe Thienard

Contents

1 Introduction **2**

2 CPUs **3**
2.1 Tiger Lake . . . 3
2.2 Ivy Bridge . . . 3

3 Filtre de Sobel 3x3 baseline **3**

4 Compilateur **4**

5 Optimisations possibles **4**
5.1 Au niveau du compilateur . . . 4
5.1.1 Les flags -O . . . 4
5.1.2 Unrolling . . . 4
5.1.3 Vectorization . . . 4
5.2 Au niveau du code . . . 6
5.2.1 Pointeur restrict . . . 6
5.2.2 Changement de type . . . 6
5.2.3 IO . . . 6
5.2.4 Inline et unroll fonction convolve_baseline . . . 9
5.2.5 Simplification de Grayscale_weighted . . . 9
5.2.6 Racine carré . . . 10

6 Filtre Sobel 7x7 **10**

7 Analyse des performances **11**

8 Conclusion **13**

1 Introduction

Le filtre de sobel est un opérateur permettant de pouvoir faire de la détection de contour. Il calcule le gradient de l'intensité de chaque pixel. Il permet d'indiquer un fort changement de contraste, passant du foncé au clair, ainsi que le changement de cette direction. Pour calculer le gradient, il nous faut utiliser deux matrices de convolution. Ces matrices nous permetteront de calculer une approximation de la dérivé horizontale et verticale du gradient, Gx et Gy. Par la suite, il nous faudra retrouver la norme du gradient à partir des approximations. Enfin, on comparera la norme, G, avec une valeur limite permettant de définir si le pixel est clair ou foncé. Résumé fait à partir de Wikipédia

$$Gx = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad Gy = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

$$G = \sqrt{Gx^2 + Gy^2}$$

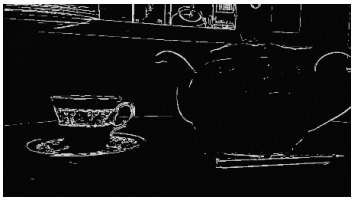
Dans nos recherches, on a pu remarquer qu'il existait plusieurs implémentations de la détection de contour comme le filtre de Canny, le filtre de Sobel ou Laplacien of Gauss. Le filtre Laplacien of Gauss est généralement plus précis qu'un filtre de Sobel mais moins rapide. Il en est de meme pour le filtre de Canny, qui quant à lui est plus encore plus précis mais bien plus lent. C'est pourquoi, on s'intéressera ici à faire un filtre de Sobel de taille 3x3 et 7x7 pour essayer de palier le manque de précision tout en gardant le maximum de performances.

Dans ce document, nous allons présenter une analyse des performances d'un benchmark sur un code Sobel. Le code sera testé sur plusieurs processeurs du fabriquant Intel. On a pu utiliser un Tiger lake ainsi qu'un Ivy bridge. Le code a été testé sur plusieurs vidéo, celle donnée de 1Go et une autre de 8Go. Les deux vidéo sont au format `.raw`. Afin de vérifier la qualité des implémentations, on a utilisé la signature sha1 sur le fichier `output.raw`. Dans un premier temps, c'est à dire sur la video de l'implémentation d'origine, nous avons obtenu : e7dcbc6f18497ab9e8bff0ca0ad0d9ce70aea245, cette signature sera changée par la suite à cause du changement de pixel gris en noir. il sera donc au final : 76636f7bddaca352d1e63696645e4a8815285ecf. Toutes les implémentations ainsi que les scripts sont disponibles sur github.

(a) Vidéo en entrée

(b) Filtre de Sobel Baseline

(c) Filtre de Sobel 3x3 sans sqrt



(d) Filtre de Sobel 7x7 sans sqrt

Le github : https://github.com/RodolpheThienard/Sobel-operator

2 CPUs

2.1 Tiger Lake

Modele	Coeurs	Fréquence GHz	L1 KiB	L2 MiB	L3 MiB	SIMD	Stockage	Vitesse lecture
Intel I7 1165g7	8	2.8	192	5	12	SSE, AVX2, AVX512	NVME	672.58 MB/sec

2.2 Ivy Bridge

Modele	Coeurs	Fréquence GHz	L1 KiB	L2 MiB	L3 MiB	SIMD	Sockage	Vitesse lecture
Intel I5 3570	4	3.4	128	1	6	SSE, AVX	SSD	209.23 MB/sec

3 Filtre de Sobel 3x3 baseline

Pour commencer, parmi le code donné, seulement les fonctions `convolve_baseline` et `sobel_baseline` seront amenées à etre modifiées. La fonction `convolve_baseline` correspond au calcul avec les deux matrices de convolution, Gx et Gy aka f1 et f2, décrites dans l'introduction.

```
i32 convolve_baseline(u8 *m, i32 *f, u64 fh, u64 fw)
{
  i32 r = 0;
  for (u64 i = 0; i < fh; i++)
    for (u64 j = 0; j < fw; j++)
      r += m[INDEX(i, j, W * 3)] * f[INDEX(i, j, fw)];
  return r;
}

void sobel_baseline(u8 *cframe, u8 *oframe, f32 threshold)
{
  i32 gx, gy;
  f32 mag = 0.0;

  i32 f1[9] = { -1, 0, 1,
        -2, 0, 2,
        -1, 0, 1 }; //3x3 matrix

  i32 f2[9] = { -1, -2, -1,
        0, 0, 0,
        1, 2, 1 }; //3x3 matrix

  for (u64 i = 0; i < (H - 3); i++)
    for (u64 j = 0; j < ((W * 3) - 3); j++)
      {
    gx = convolve_baseline(&cframe[INDEX(i, j *3, W * 3)], f1, 3, 3);
    gy = convolve_baseline(&cframe[INDEX(i, j *3, W * 3)], f2, 3, 3);
    mag = sqrt((gx * gx) + (gy * gy));
    oframe[INDEX(i, j, W * 3)] = (mag > threshold) ? 255 : mag;
      }
}
```

4 Compilateur

Dans un premier temps, on a décidé de comparer les différents compilateurs. On a utilisé les trois compilateurs suivants, GCC, ICX et CLANG. On a pu remarquer avec les premières prises de mesures, sur le code avec l'implémentation donnée, qu'il y avait des disparités entre les trois compilateurs. On peut s'attendre à avoir deux groupes dans les performances mesurées. De l'un comptant ICX et CLANG qui sont tous les deux des compilateurs LLVM et de l'autre, GCC. Ci-dessous, vous trouverez les différentes versions des compilateurs.

Compilateur	**Version**
ICX	2022.1.0.20220316
GCC	12.2.1
CLANG	15.0.7

5 Optimisations possibles

5.1 Au niveau du compilateur

Dans cette section, nous parlerons des optimisations faites par le processeur lors de la compilation. Un flag important de mettre lors de la compilation est `-march`. Ce flag permet au compilateur d'appliquer les optimisations propre à l'architecture. Ce flag peut etre utilisé de facon générique en mettant `-march=native` comme argument.

Un second flag important lors de la compilation est `-finline-functions`. Il permet au compilateur d'effectuer une extension de fonction en ligne pour les appels aux fonctions définies dans le fichier source actuel.

5.1.1 Les flags -O

Le flag `-O1` permet aux compilateurs de mettre un ensemble de flags permettant de reduire la taille du code et le temps d'exécution. Il ne fait aucune autre optimisation en particulier.

Avec le flag `-O2`, le compilateur applique des optimisations de `-O1` tout en poussant la performance du code généré en inlinant les fonctions ou en les alignant. Le flag `-O2` applique également des optimisations sur la vectorisation par default. Si la possibilité de vectorisation sur une boucle est possible, le compilateur la génèrera automatiquement.

Le flag `-O3` applique toutes les optimisations des deux précédents en insistant sur la vectorisation des boucles en se permettant l'interchangage de boucle.

Pour `-Ofast`, les changements par rapport à `-O3` sont des optimisations reduisant la précision mathématique, avec `-ffast-math`. Son but est de réduire la précision au profit de la performance.

5.1.2 Unrolling

Le flag `-funroll-loops` permet au compilateur de dérouler automatiquement une boucle quand cela est possible. Ce paramètre permet un meilleur déroulage que s'il était fait par le développeur mais également rend l'entretient plus simple.

5.1.3 Vectorization

Un flag manquant pour la vectorisation est `-ftree-vectorize`. Il permet une vectorisation en arbre et active deux flags présents à partir de `-O2`.

Il peut etre nécessaire de rajouter des flags utiles memes si ils sont présent dans les flags `-O`. Cela permet d'essayer de les réappliquer s'il y a eu un échec avant.

Version non optimisée : `-march=native -O1`
Version optimisée : `-march=native -Ofast -funroll-loops -ftree-vectorize -finline-functions -qmkl`

Compilateur	Version non optimisée Tiger Lake	Version optimisée Tiger Lake	Version non optimisée Ivy Bridge	Version optimisée Ivy Bridge
ICX	150 MiB/s	2047 MiB/s	111 MiB/s	456 MiB/s
GCC	179 MiB/s	1980 MiB/s	92 MiB/s	645 MiB/s
CLANG	205 MiB/s	2063 MiB/s	109 MiB/s	868 MiB/s

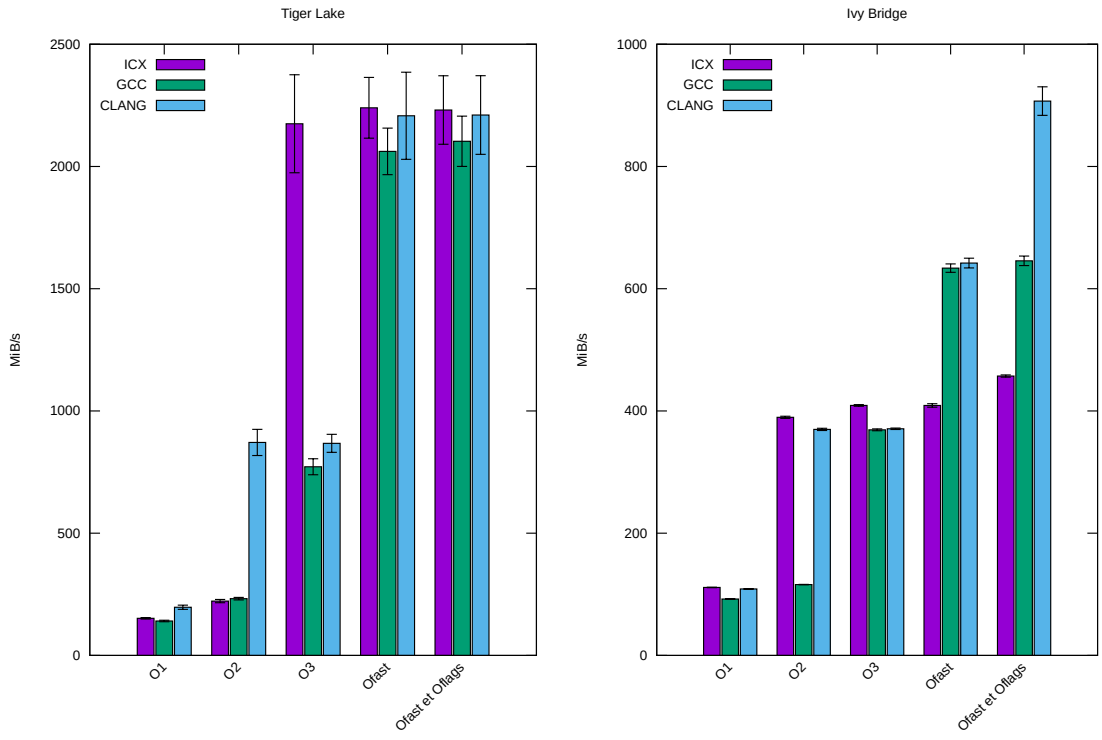


Figure 2: Comparaison des optimisations CPU en fonctions flags

On a pu remarquer un très net écart entre `O2` et `O3` pour ICX sur Tiger Lake Cela doit etre du au fait que ICX applique des flags à partir de `O3` alors que GCC et CLANG eux ne les appliquent qu'à partir de `Ofast`. Pour ce qui est de Ivy Bridge, on remarque un très net speed up avec CLANG pour la version complètement optimisée. Nous nous attendions à voir ICX au moins au meme niveau que CLANG, on est surpris.

5.2 Au niveau du code

5.2.1 Pointeur restrict

Ajout de l'argument `restrict` pour tous les pointeurs de données. L'argument `restrict` permet de dire au compilateur que les pointeurs passés à cette fonction ne se recouvrent pas. Cette modification au niveau du code n'est pas une optimisation pour obtenir un gain de performance.

5.2.2 Changement de type

On s'est également intéressé aux types des variables. On s'est demandé si ils étaient tous de la bonne taille. On a pu remarquer que les matrices de convolution sont écrites dans des tableaux de taille 32bits, or elles n'utilisent que cinq valeurs (-2,-1,0,1,2). On a donc décidé de changer le type i32 en i8. De meme pour les gradients, anciennement stocké en i32, ils ne peuvent monter qu'à 2^16. Il est donc suffisant d'utiliser un i16. Nous avons calculé un gain de performance de **1.13** en moyenne.

```
i32 gx, gy;
i32 f1, f2;

// After optimisation
i16 gx,gy;
i8 f1, f2;
```

5.2.3 IO

On a essayé plusieurs implémentations pour la partie IO. En allouant le ficher `.raw` avec nmap, mais également en remplaçant `_mm_malloc` par `aligned_alloc` ou `malloc`. `malloc` est une fonction permettant d'allouer de la mémoire. Cette mémoire n'est ni initialisé, ni alignée. `aligned_alloc` et `_mm_malloc` permettent également d'allouer de la mémoire. Cependant, ils prennent un paramètre supplémentaire qui est une contrainte d'alignement. La mémoire est garantie d'etre alignée dans cette limite spécifiée.

Après toutes nos tentatives, on a pu conclure que `_mm_malloc` était la meilleure implémentation parmi nos tentatives.

Pour la partie IO, on a voulu savoir quel était l'impact des Input/Ouput sur les performances du code. On a donc regardé l'évolution du temps d'exécution en fonction des implémentations. Nous avons remarqué que les deux étaient liés mais pas autant qu'on peut le penser. Les couts IO ne sont pas négligeables.

On a donc décidé d'implémenter `aligned_alloc` et `malloc` pour voir si une optimisation serait possible. On a pu remarquer à travers les graphs que l'implémentation avec `_mm_malloc` est plus efficace que les deux autres testées pour ce qui est du temps total. De plus il y a une forte disparité entre les deux CPU testés au point de vu du calcul par image avec très peu de variations pour l'architecture Ivy Bridge.

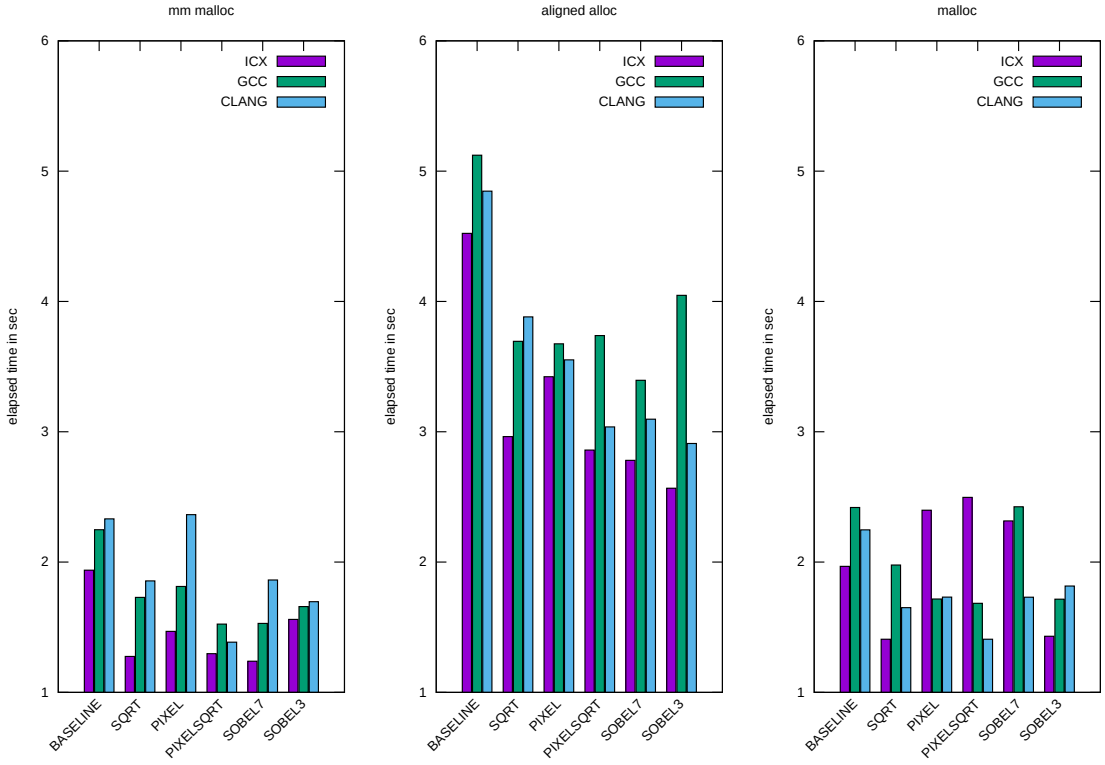


Figure 3: Comparaison de l'évolution du temps total sur Tiger Lake

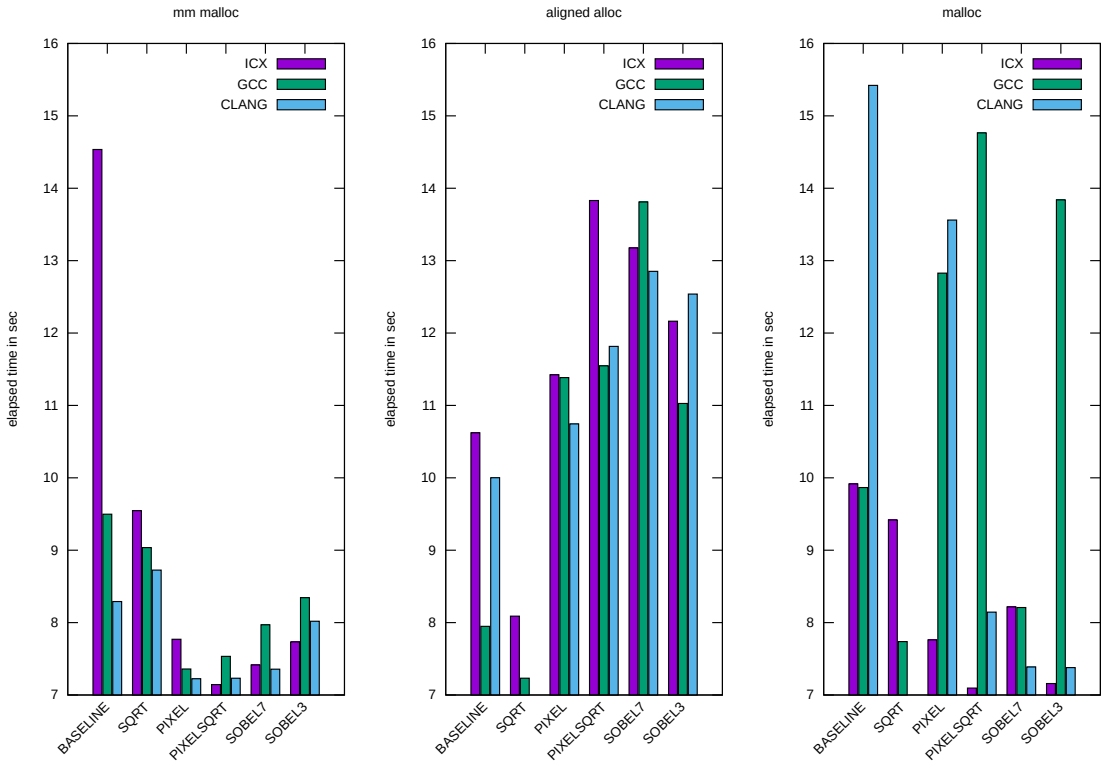


Figure 4: Comparaison de l'évolution du temps total sur Ivy Bridge

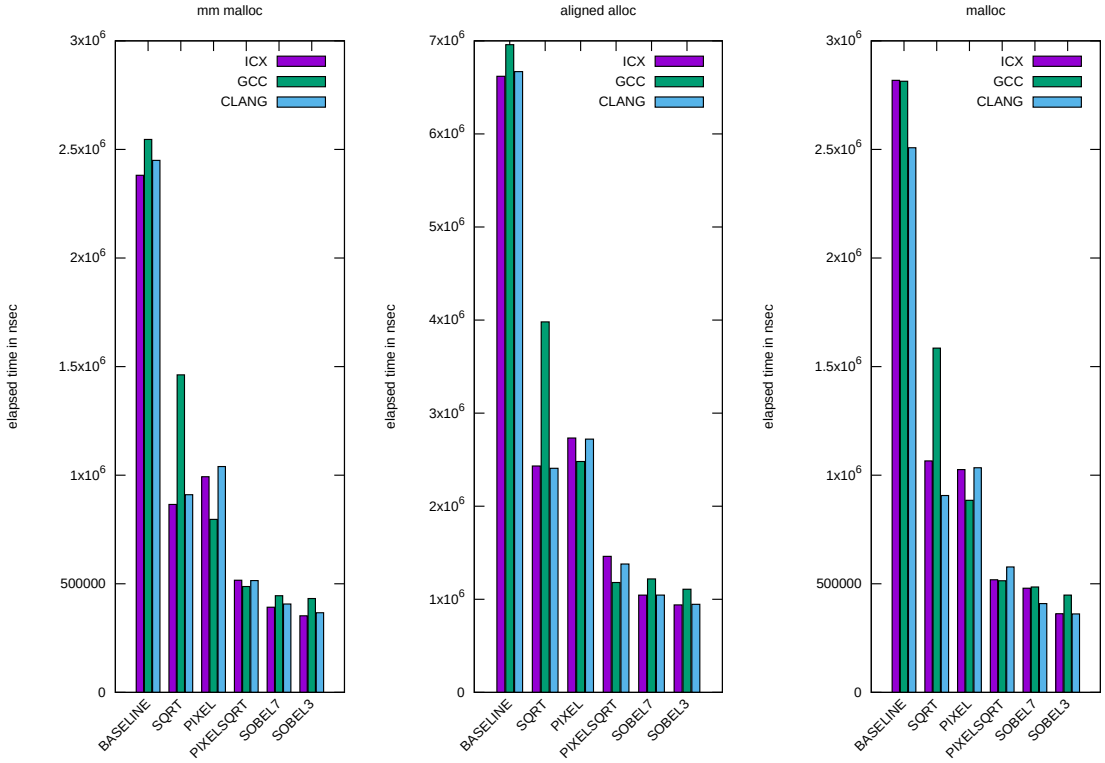


Figure 5: Comparaison de l'évolution du temps par image sur Tiger Lake

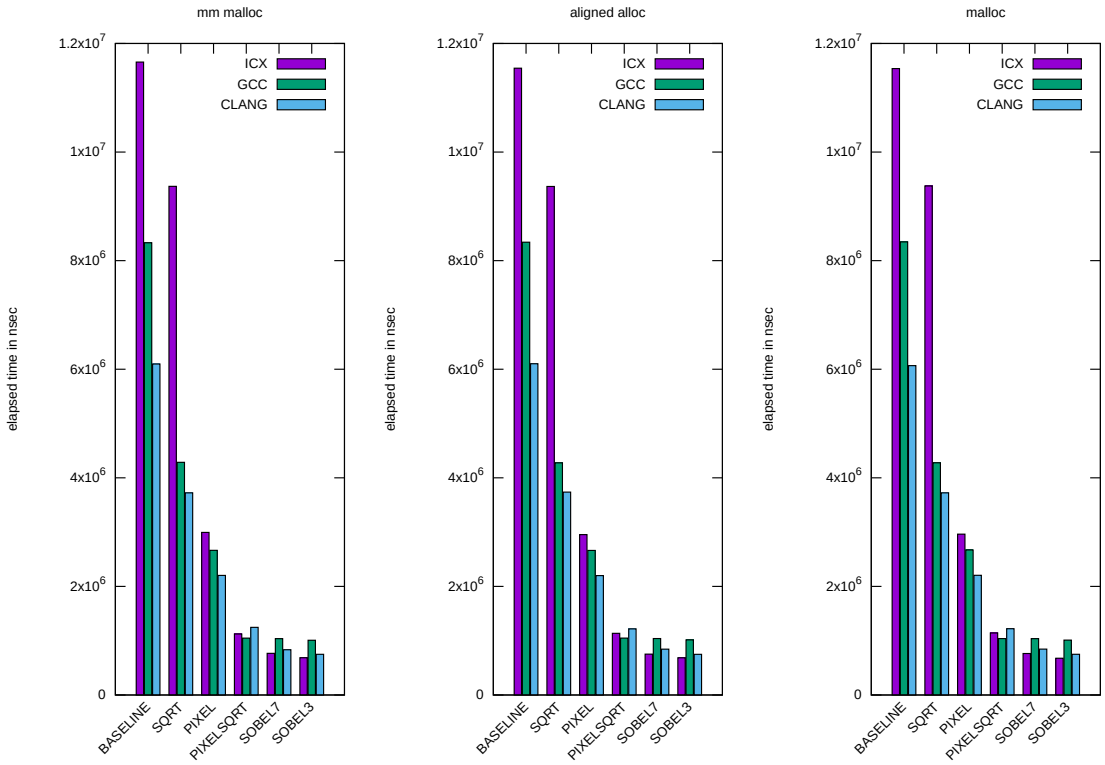


Figure 6: Comparaison de l'évolution du temps par image sur Ivy Bridge

5.2.4 Inline et unroll fonction convolve_baseline

Pas de gain de performance notable. Les flags de compilation étant `-funroll-loops -ftree-vectorize -finline-functions`, on peut penser que le compilateur avait déjà appliqué ces modifications. En regardant les matrices f1 et f2, on remarque qu'elles ont 3 valeurs à zéro. On a donc décidé de ne pas les calculer. Les matrices étant écrites dans le code, le compilateur, lors du unroll, doit également supprimer ces itérations. C'est pourquoi, meme après ce changement, nous ne remarquons aucun gain de débit.

```
// Convolve
for (u8 ii = 0; ii < 3; ii++) {
    gx += m[INDEX(ii, 0, W)] * f1[INDEX(ii, 0, 3)];
    gx += m[INDEX(ii, 2, W)] * f1[INDEX(ii, 2, 3)];

    gy += m[INDEX(0, ii, W)] * f2[INDEX(0, ii, 3)];
    gy += m[INDEX(2, ii, W)] * f2[INDEX(2, ii, 3)];
}
```

5.2.5 Simplification de Grayscale_weighted

Nous avons pu remarquer que la fonction grayscale parcourait H en modifiant les trois composantes de couleurs (RVB) par une meme valeur. C'est pourquoi, on a décider de remplacer seulement la valeur i et non plus i, i+1 et i+2. Dans la suite, pour réécrire les composantes RVB, nous avons juste réécrit trois fois la meme. Ce changement permet de ne calculer qu'un tier des valeurs étant donné que la matrice ne comporte qu'un gris au lieu de trois. Le gain calculé est très surprenant et meme supérieur à nos attentes.

Gain	Calculé Tiger Lake	Calculé Ivy Bridge	Théorique
ICX	x2.63	x3.97	x3
GCC	x3.18	x3.16	x3
CLANG	x2.70	x2.87	x3

```
void grayscale_weighted(u8 *frame)
{
    f32 gray;
    for (u64 i = 0; i < H * W * 3; i += 3) {
        gray = ((float)frame[i] * 0.299) +
               ((float)frame[i + 1] * 0.587) +
               ((float)frame[i + 2] * 0.114);
        frame[i/3] = gray;
    }
}
// Updating loop in sobel_baseline
    for (u64 i = 0; i < (H - 3); i++)
        for (u64 j = 0; j < ((W ) - 3); 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);

            mag = sqrt((gx * gx) + (gy * gy));
            u8 gray = (mag > threshold) ? 255 : mag;
            oframe[INDEX(i, j*3, W * 3)] =  gray;
            oframe[INDEX(i, j*3+1, W * 3)] =    gray;
            oframe[INDEX(i, j*3+2, W * 3)] =    gray;
}
```

5.2.6 Racine carré

La racine carré est une opération complexe pour le processeur, coutant au minimum 12 cycles. Elle peut entrainer des bottlenecks. Sur les versions de Sobel avec l'opération `SQRT`, racine carré, nous avons un bottleneck augmentant le coup en cycle par itération. Celui-ci étant à 96 cycles peut etre réduit à 66 cycles sans les bottlenecks. (Utilisation de maqao pour connaitre les métriques).

Dans la recherche d'optimisation de l'algorithme Sobel, nous nous sommes permis de changer quelques valeurs données dans le code de base. Les valeurs changées auront un impact très faible sur la détection de contour. Pour commencer, on a retiré, à l'écriture, la valeur `mag`. Celle-ci sera remplacée par 0 afin de pouvoir augmenter le contraste qui sera maintenant noir et blanc mais surtout pour permettre de faire un important gain de performance.

Pour toutes les implémentations n'ayant plus la racine carré, la signature du fichier sera différente, comme signalé dans l'introduction.

La signature sans la racine carre : 76636f7bddaca352d1e63696645e4a8815285ecf

Gain	SQRT	Sans SQRT Tiger Lake	Sans SQRT Ivy Bridge
ICX	x1	x2.64	x1.27
GCC	x1	x1.73	x1.96
CLANG	x1	x2.68	x2.28

```
u32 threshold = 100.0;
mag = sqrt((gx * gx) + (gy * gy));
u32 gray = (mag > threshold) ? 255 : mag;

// After removing square root
u32 threshold = 10000.0;
mag = (gx * gx) + (gy * gy);
u32 gray = (mag > threshold) ? 255 : mag;
```

6 Filtre Sobel 7x7

Nous avons pu lire également dans ce projet un rapport sur les sobels. Ce rapport, *Custom Extended Sobel Filters* , nous apprend qu'il est possible d'obtenir une meilleure définition du contour en utilisant un filtre de sobel de taille 7x7. Il y est également mentionné que le filtre de taille 7x7 est le plus optimal en temps/qualité.

C'est pourquoi, nous nous sommes lancés dans la modification du code de telle sorte à utiliser le filtre 7x7 à partir de la version la plus optimale avec un filtre 3x3.

Comparaison des performances entre un filtre de sobel 3x3 et 7x7

Version Optimale	Tiger Lake GiB/s	Ecart-type	Ivy bridge GiB/s	Ecart-type
ICX Filtre 3x3	14	9.6%	7.8	5.9%
ICX Filtre 7x7	12.8	9%	7.0	5.6%
GCC Filtre 3x3	12	9.3%	7.0	5.5%
GCC Filtre 7x7	11.3	6.7%	5.1	4%
CLANG Filtre 3x3	13.7	9.5%	7.0	5.3%
CLANG Filtre 7x7	12.7	8.5%	6.3	4.8%

7 Analyse des performances

Les versions mesurées sur Tiger Lake et sur Ivy Bridge ne seront pas comparées graphiquement étant donné qu'elles sont chacune comparées à leur version non optimisées qui est différente suivant le compilateur utilisé et l'architecture. De plus, sur chaque graphique, les versions montrent le speed up entre la version non optimisée et la version testée suivant le compilateur.

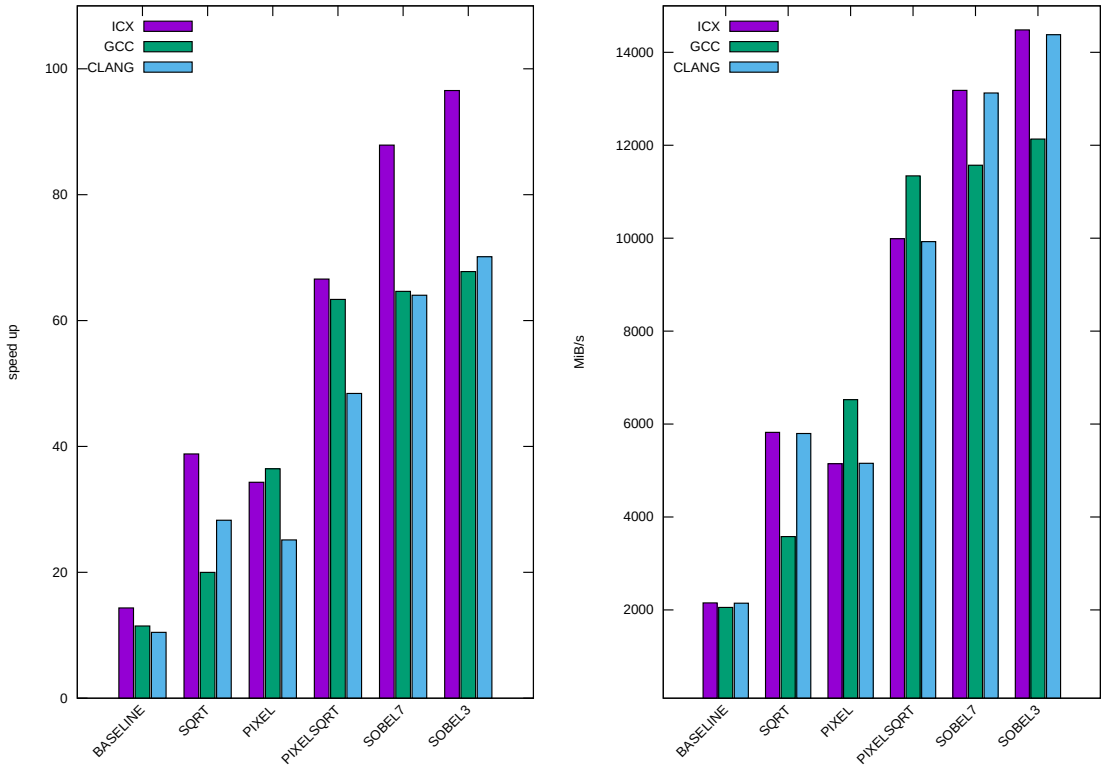


Figure 7: Comparaison des implémenations sur chaque compilateur sur Tiger Lake

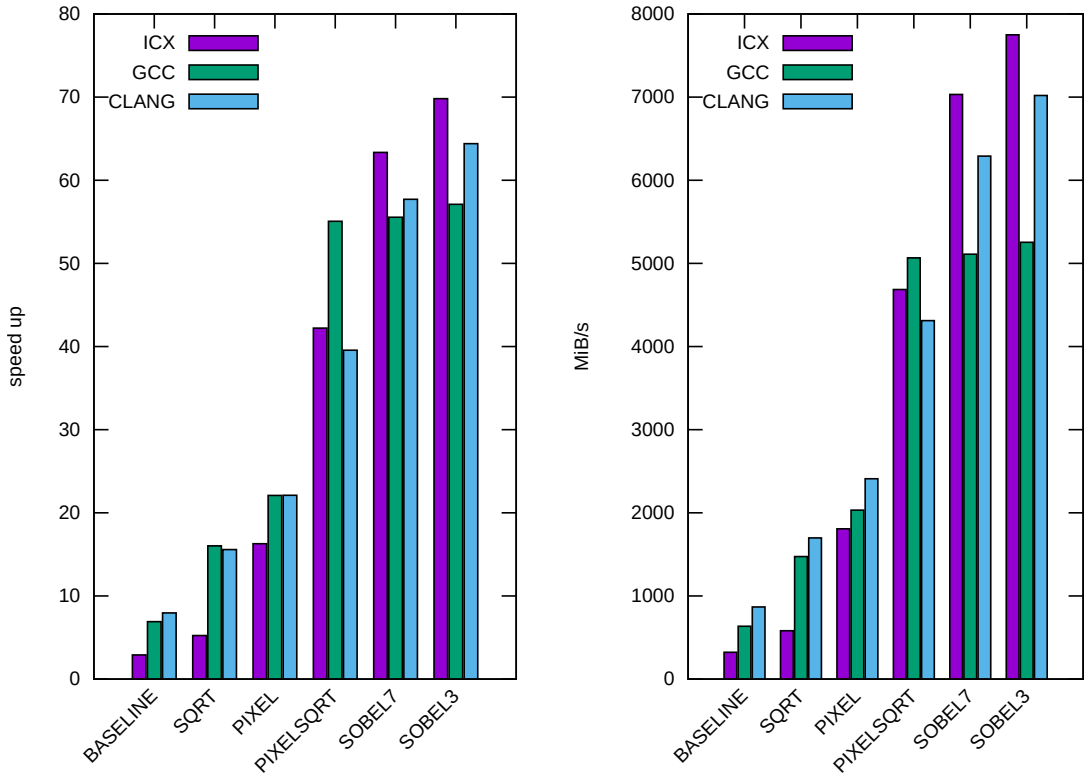


Figure 8: Comparaison des implémenations sur chaque compilateur sur Ivy Bridge

8 Conclusion

Parmi le travail effectué, plusieurs implémentations ont permis d'améliorer l'efficacité du calcul du filtre de Sobel. On a, à travers elles, pu voir les paramètres prédominants dans son temps d'exécution. La racine carré est l'un des paramètres les plus importants couplé avec le calcul du grayscale. Eux deux représentent, plus de 3/4 des gains obtenus grace à la modification du code. De plus, les gains engendrés par les flags de compilations ne sont pas négligeables et imposent au développeur de les choisir rigoureusement.

Le travail fourni ne couvre cependant pas toutes les améliorations possible pour un filtre de Sobel. En amélioration envisageable dans le futur, on peut penser à paralléliser le code, avec thread ou OpenMP. Cela permettrait de faire un important gain sur les appels IO, qui sont actuellement le plus couteux. Une fois ce code en parallèle, on pourrait également essayer une implémentation Sycl de OneAPI développé par Intel afin d'essayer sur un GPU. Il y a enfin la possibilité d'améliorer la vectorisation pour que le code puisse entièrement utiliser les registres Zmm sur une architecture disposant de l'AVX512.