Analyse de performances d'un benchmark de filtre de Sobel

Pacôme Rousselle

21 Janvier 2023

https://github.com/Pacome-Rousselle/M1_CHPS_AP_ROUSSELLE_SOBEL

Table des matières

1 Introduction

Ce document présente une analyse des performances d'un filtre de Sobel, benchmark sur l'architecture x86_64.

Cette analyse présentera dans l'ordre l'implémentation de base du filtre de Sobel, l'architecture cible, les différentes optimisations appliquées (au niveau du code et de la compilation) et enfin les mesures de performance réalisées pour chaque compilateur.

1.1 Filtre de Sobel

En HPC, des kernels de convolution sont souvent utilisés pour détecter les contours d'une image. Il en existe 4 variantes, chacun pour un contour :

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

De gauche à droite : top, left, right et bottom filter.

On le retrouve fréquemment en HPC, notamment par l'emploi d'opérations matricielles, de calcul de gradient, ou encore par sa présence dans le traitement d'image pour réseaux de neurones.

Dans le cas d'un filtre de sobel, soit le vecteur gradient $G(G_x, G_y)$, avec A la partie 3x3 de l'image (en niveaux de gris) traitée par convolution :

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} A \quad \text{et} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} A$$

Le pixel i de la nouvelle image traitée prend la valeur de la norme de G_i:

$$||G_i|| = \sqrt{G_{ix}^2 + G_{iy}^2}$$

2 Implémentation de base

Le code ci-dessous applique le filtre de sobel sur un pointeur de char et écrit dans un autre pointeur de char.

```
//
for (u64 i = 0; i < (H - 3); i++)
  for (u64 j = 0; j < ((W * 3) - 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));

      oframe[INDEX(i, j, W * 3)] = (mag > threshold) ? 255 : mag;
    }
```

Avec l'appel à convolve_baseline ci-dessous. L'algorithme de convolution nécessite deux chargements mémoires et deux opérations arithmétiques (add & mul) par itération. Ici fh = fw = 3, et on effectue 3x3 itérations. La complexité de l'algorithme est donc en $O(n^2)$.

```
i32 r = 0;

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

Sachant que les données sont accédées de manière linéaire (et déclarées en ligne avec _mm_malloc) avec un pas constant de 1, le pattern de ce code peut profiter de certaines caractéristiques du CPU telles que le SIMD (Single Instruction Multiple Data, pour de la vectorisation) et le prefetching de données.

Le kernel effectuant également une addition et une multiplication combinées à chaque itération, les unités MCA/FMA (Multiply-Accumulate/Fused-Multiply-Add) présentes dans la plupart des architectures modernes peuvent donc apporter des réductions du temps d'exécution.

3 Matériel employé

3.1 Architecture cible

Les mesures ont été réalisées sur un AMD Zen 2, une architecture x86_64 (les tailles de caches ne sont pas renseignées au vu de la taille du problème, qui ira toujours en RAM):

Modèle	Coeurs	Fréquence en GHZ	SIMD
Ryzen 5 PRO 4650U - AMD	6	2,1	SSE, AVX2

3.2 Benchmarking

Les implémentations de la fonction sobel seront passées dans une fonction "run_benchmark", qui retournera des informations comme les temps minimaux, maximaux et moyens de chaque exécution, ainsi que la bande passante et son écart-type.

3.2.1 Compilateurs

Le code est compilé avec gcc 11 et clang 14, chacun avec le flag de compilation -03 pour ses capacités de vectorisation. On s'attend à une compilation plus rapide avec clang.

4 Optimisations possibles

4.1 Au niveau du code

Cette section couvre toutes les modifications à apporter dans le code à la fonction sobel_baseline pour améliorer les performances du programme. Il faut noter que chaque version aura ces pointeurs avec un check d'alignement mémoire *restrict*.

4.1.1 Déroulage à 3

Une première optimisation consiste au déroulage de la fonction de convolution. Cette transformation consiste en la duplication du corps de la boucle intérieure, afin de réduire le nombre d'itérations (ici, de 9 à 3) et d'augmenter la charge de travail du CPU pour une itération, en approchant le plus possible de son maximum. Ceci peut augmenter les performances car la plupart des architectures modernes implémentent une prédiction de branchement (prédiction de quelles instructions seront réalisées dans une structure de boucle ou de if-else avant leur appel) et du "heavy pipelining" (augmentation du nombre d'instructions pouvant être réalisées en même temps).

Ci-dessous, le déroulage à 3 de la fonction de convolution :

```
i32 convolve_unroll(u8 *m, i32 *f, u64 fh, u64 fw)
{
  i32 r = 0;

  for (u64 i = 0; i < fh; i++)
  {
    r += m[INDEX(i, 0, W * 3)] * f[INDEX(i, 0, fw)];
    r += m[INDEX(i, 3, W * 3)] * f[INDEX(i, 1, fw)];
    r += m[INDEX(i, 6, W * 3)] * f[INDEX(i, 2, fw)];
  }
  return r;
}
```

La principale difficulté rencontrée avec le déroulage est que le nombre d'éléments sur lequel on travaille n'est pas toujours divisible par le facteur de déroulage (exemple : un déroulage à 4 sur un tableau de 7 éléments). Il faut alors ajouter un "collecteur de miettes", une boucle s'occupant du reste des éléments à travailler. Nous avons évité ce problème ici, en choisissant un facteur 3 sur des tableaux de 9 éléments, permettant de supprimer une boucle en appliquant le filtre de sobel ligne par ligne.

4.1.2 Elimination des éléments à 0

Les opérations impliquant des produits nuls coûtent en temps d'éxécution. C'est le cas d'un tiers des éléments des deux kernels employés, entraînant donc une opération inutile sur trois dans la fonction de convolution (right sobel), jusqu'à une itération entièrement inutile dans le cas du bottom sobel. En plus d'une perte de temps, il y a une perte d'espace, car il faut stocker les 0.

On se propose donc remplacer la déclaration actuelle des tableaux f1 et f2 en supprimant leurs éléments nuls, faisant passer leur taille de 9 à 6 (3x2 et 2x3 respectivement).

Ci-dessous, les deux nouvelles fonctions de convolution, accédant séparément aux éléments des filtres :

```
i32 convolve_right(u8 *restrict m, i32 *restrict f, u64 fh, u64 fw)
{
  i32 r = 0;

  for (u64 i = 0; i < fh; i++)
  {
    r += m[INDEX(i, 0, W * 3)] * f[INDEX(i, 0, fw)];
    r += m[INDEX(i, 6, W * 3)] * f[INDEX(i, 1, fw)];
  }

  return r;
}
```

```
i32 convolve_bottom(u8 *restrict m, i32 *restrict f, u64 fh, u64 fw)
{
  i32 r = 0;
  u64 invariant;

  for (u64 i = 0; i < fh; i++)
  {
    invariant = i*fh;
    r += m[INDEX(i, 0, W * 3 *invariant)] * f[INDEX(i, 0, fw)];
    r += m[INDEX(i, 3, W * 3 *invariant)] * f[INDEX(i, 1, fw)];
    r += m[INDEX(i, 6, W * 3 *invariant)] * f[INDEX(i, 2, fw)];
  }

  return r;
}
```

Les boucles ont été déroulées pour accéder au mieux aux éléments des tableaux (ce n'était pas nécessaire pour convolve_bottom, mais impossible autrement pour convolve_right avec l'incrémentation de j).

4.1.3 Suppression de sqrt

Une opération très coûteuse en temps d'exécution est la fonction sqrt, nécessaire pour calculer la norme du gradient. Une première idée a été d'appeler une fonction BLAS, srnm2, afin de directement calculer la norme du gradient. Cela s'est révélé plus coûteux encore, l'appel à BLAS n'étant pas rentable en temps pour un vecteur de taille 2.

L'idée retenue est une approximation : après le calcul de la norme de gradient, on la compare à une limite (threshold) pour déterminer la valeur du pixel final. L'approximation choisie est que si on compare $\sqrt{G_{ix}^2 + G_{iy}^2}$ à threshold, alors on peut comparer $G_{ix}^2 + G_{iy}^2$ à $threshold^2$ au vu des propriétés de conservation de la fonction carrée (à la condition qu'en tout point on obtienne $|a| < |b|$ pour avoir $a^2 < b^2$).

Les seuls changements apportés à la fonction baseline sont les suivants :

```
mag = (gx * gx) + (gy * gy);
threshold *= threshold;
```

La vidéo produite montre que cette approximation est juste, on ne perd pas en détection de contour.

4.2 Au niveau du compilateur

Cette section couvre toutes les modifications à apporter à la commande de compilation du code pour adapter au mieux le code à l'architecture et ce pour améliorer les performances du programme. On emploiera ici les compilateurs gcc et clang.

4.2.1 Flags de compilation

Un des flags de compilation les plus importants est -march=native, qui spécifie l'architecture, ce qui aide le compilateur à appliquer les optimisations disponibles et adaptées au code. D'autres flags comme -02, -03 et -Ofast pour gcc et clang peuvent ajouter des optimisations au temps d'éxécution. Dans notre cas on a choisi d'utiliser seulement -03, car il inclut -02 et qu'Ofast n'est intéressant que lors d'utilisation de calcul flottant.

4.2.2 Déroulage

Le flag -funroll-loops renseigne au compilateur que l'on cherche à dérouler les boucles, si cela est possible, ce qui permet de ne pas à le faire soi-même au niveau du code et donc évite des erreurs humaines dans le déroulage, notamment dans les indices des matrices.

4.3 Vectorisation

Un autre moyen d'améliorer la bande passante est la vectorisation, qui augmente le nombre de données traitées en même temps. Cette section présente les optimisations possibles à ce niveau.

4.3.1 A la compilation

Le compilateur peut gérer de lui-même la vectorisation, comme par exemple gcc 12.2 avec le flag -O2, qui vectorise des boucles quand il le peut.

On peut également placer le pragma omp simd au-dessus d'une boucle, pour créer une région parallèle indiquant à la compilation le partage de la boucle en threads. Cette optimisation n'a pas montré de résultats probants malgré différents essais, soit parce que les arguments entrés ne sont pas bons, soit que la boucle n'est peut-être pas adaptée à cette directive.

4.3.2 Dans le code

Comme dit en début de section et en introduction, la mémoire est accédée de manière linéaire dans nos boucles, ce qui pourrait permettre d'appliquer les fonctionnalités SIMD du CPU, résultant alors en une meilleure bande passante. Cette vectorisation est possible par l'appel d'asm, permettant d'insérer du code assembleur dans un code C

A l'heure de l'écriture de ce rapport, nous n'avons pas pu implémenter de telles fonctions, mais l'emploi du set d'instruction AVX a été envisagé. Cependant, la taille des registres xmm (128 bits) pose problème (128 bits correspondent à 4 ints, posant des problèmes d'indices de matrice avec l'organisation 3 par 3). Une utilisation des registres ymm (256 bits, soit 8 int) semble être plus pertinente. On peut également ajouter qu'AVX contient des instructions MAC/FMA, utiles notamment pour le calcul de la norme du gradient. L'utilisation d'intrinsics Intel peut également remplacer l'appel à asm pour une syntaxe simplifiée et une meilleure portabilité du code.

5 Comparaison de performances

Cette section présente les résultats obtenus après avoir fait tourner les différentes implémentations du kernel sur l'architecture citée en section 3, l'AMD Zen 2. Comme métrique de performance nous avons choisi la bande passante en MiB/s. La version de base, telle que donnée au début du projet, est en rouge sur les graphiques.

5.1 Avec gcc

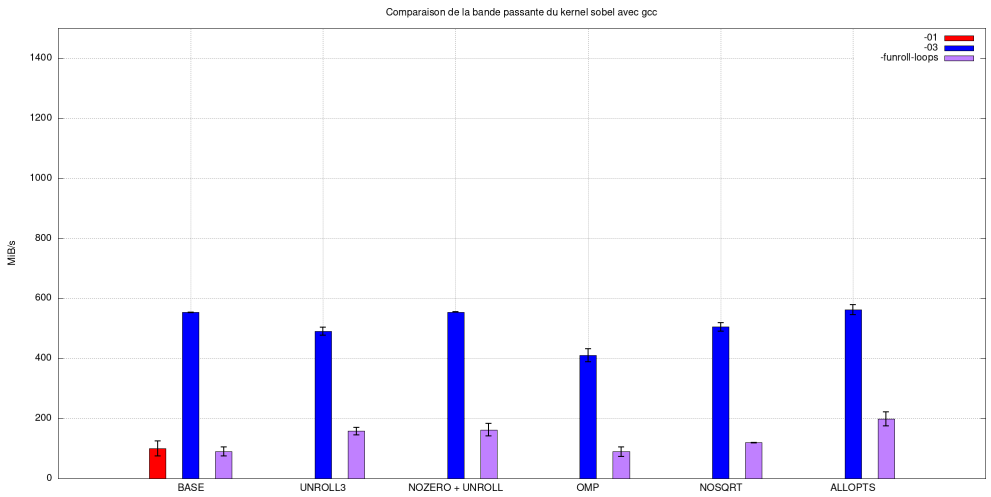

Figure 1: Comparaison des performances de différentes implémentations de sobel, compilées avec gcc

Sur le graphique nous pouvons constater que l'amélioration individuelle de chaque optimisation au niveau du code est peu significative, les principales améliorations étant du fait des flags.

5.2 Avec clang

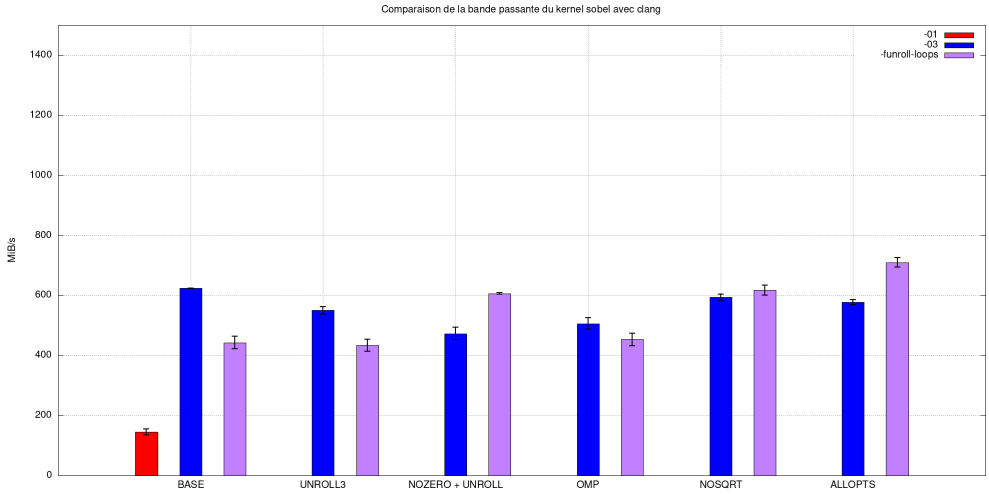

Figure 2: Comparaison des performances de différentes implémentations de sobel, compilées avec clang

Les mêmes observations qu'avec gcc peuvent être faites, cependant on peut noter une meilleure bande passante en moyenne avec clang, du fait des spécificités de ce compilateur, notamment pour le déroulage de boucles avec -funroll.

5.3 Annexe

Si les résultats du code ne sont pas concluants par rapport aux flags -03 et -funroll, on constate pour les deux compilateurs une amélioration pour les flags -01 et -02 :

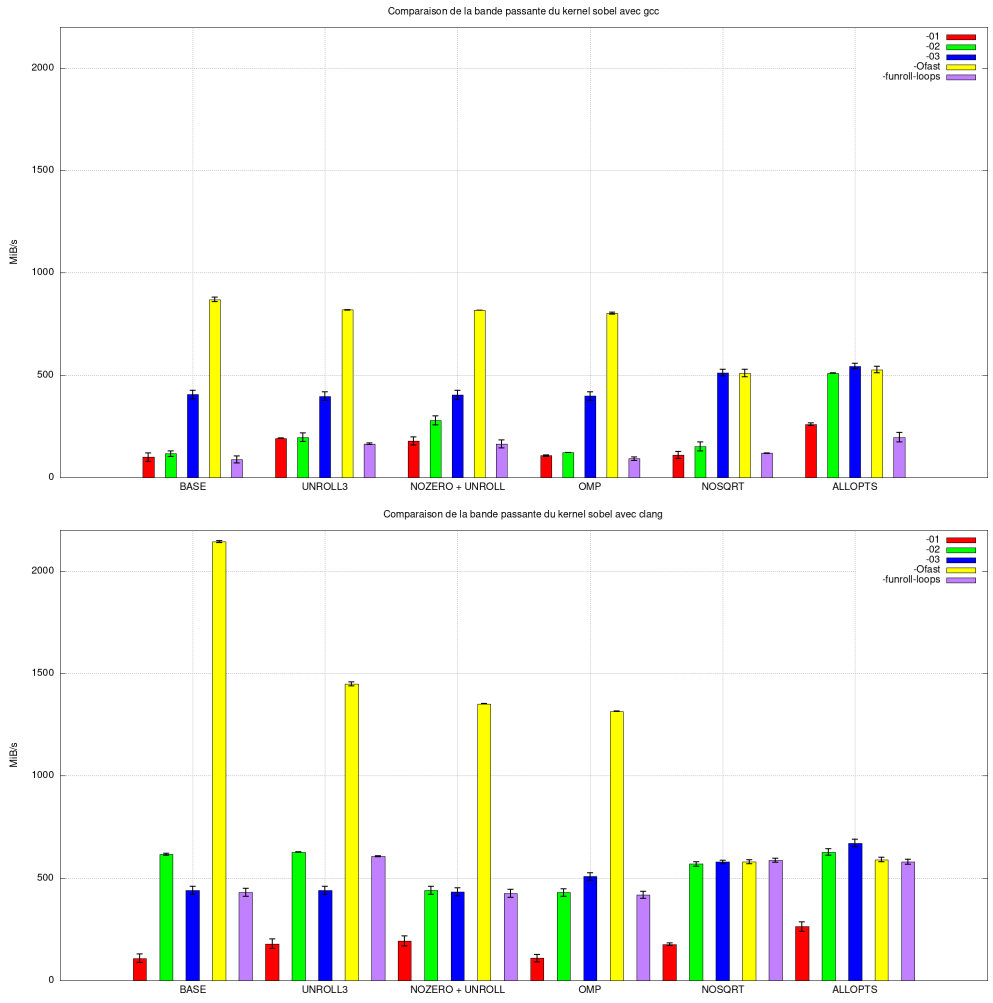

Figure 3: Mesures précédentes, incluant plus de flags de compilation

6 Conclusion

Dans ce document nous avons présenté différentes implémentations du filtre de sobel ainsi que de possibles optimisations : déroulage, vectorisation, parallelisation, compilation, afin d'en améliorer la performance sur différentes architectures en utilisant gcc et clang.

Nous avons également présenté les résultats de performance mesurés sur un AMD Zen 2 (Ryzen 5 PRO 4650U). A l'observation des résultats, nous concluons que la compilation sous clang avec le flag -funroll-loops sur le code avec toutes les optimisations du code implémentées est la version la plus efficace.

6.1 Améliorations possibles

- Compiler le programme avec plus de compilateurs :
  Aocc de AMD, icc et icx pour Intel.

- Implémenter des versions en assembleur *inline* : SSE, AVX, AVX512.

- Améliorer la qualité de la compilation : Des pertes de performance sont peut-être le fait de concurrence d'optimisation (i.e, on applique une optimisation dans le code et dans la compilation, ce qui n'est pas nécessaire).

- Ajouter de nouvelles métriques, comme le nombre de frames traitées en une seconde.

6.2 Remerciements

M. "yaspr" Ibnamar, pour l'exemple de rapport de performance utilisé pour l'écriture de ce document.

Thomas Roglin, propriétaire de la machine sur laquelle les mesures ont été prises.

Gabriel Dos Santos, pour sa relecture du document.