Université de Versailles Saint-Quentin-en-Yvelines

Calcul Haute Performance & Simulation

Projet d'Architecture Parallèle

Tom Budon

Table des matières

1 Introduction **2**

2 Kernel **2**

3 Architecture cible **4**

4 Optimisation **4**
4.1 Déroulage de boucle 4
4.1.1 Déroulage de boucle avec le compilateur 4
4.1.2 Déroulage de boucle manuel 4
4.2 Vectorisation 6
4.2.1 Compilateur Auto-Vectorisation 6
4.2.2 Intrinsics 6
4.3 Alignement Mémoire 8
4.4 Instructions coûteuses 8
4.5 Parallélisation 9

5 Résultats **10**

6 Conclusion **12**

1 Introduction

Dans ce document, je vais présenter les optimisations que j'ai effectué sur un code de détection des contours grâce à un filtre de sobel et une convolution.

2 Kernel

Le code donné si-dessous applique un filtre de sobel à une image, avec pour but de détecter les contours des formes dans l'image.

Le code de la convolution, qui applique un filtre sur une matrice, a une complexité en O(1), parce que la convolution sera toujours appelée avec un filtre de sobel de 3x3 et ne fera que 9 itérations. Cet algorithme nécessite 2 load, 1 add et 1 mul par itération.

```
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;
}
```

0.1 – Implémentation de la convolution

Le code suivant va appliquer un filtre de sobel (en x et en y) à une image. Il a une complexité en $O(n^2)$, avec 1 store, 1 square root, 2 mul, 1 add et 2 appels à la convolution par itération.

```
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, 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;
    }
  }
}
```

0.2 – Implémentation de sobel

Ici nous allons utiliser ce code sur une vidéo avec 360 frames, on appliquera donc le filtre de sobel consécutivement sur les 360 images de la vidéo.

3 Architecture cible

Model	Cores	Frequency GHz	CacheLine size Bytes
Intel(R) Core(TM) i3-6006U	2	2.0	64

L1 size KiB	L2 size KiB	L3 size MiB	SIMD
32	256	3	SSE, SSE2, AVX, AVX2

4 Optimisation

4.1 Déroulage de boucle

Le déroulage de boucle minimise les branchements dans le code cela sert à optimiser les calculs du CPU, il passera plus de temps à faire des calculs et moins de temps à faire des JUMP.

4.1.1 Déroulage de boucle avec le compilateur

Avec le flag -funroll-loops le compilateur essaiera de dérouler les boucles si il le peut.

4.1.2 Déroulage de boucle manuel

Le code si-dessous duplique le corps de la boucle 4 fois (coupé dans l'image), de plus les appels à la convolutions on été supprimé et celle-ci est aussi déroulé (en supprimant les multiplications par 0).

```
void sobel_unroll4(u8 *cframe, u8 *oframe, f32 threshold)
{
  i32 gx = 0, gy = 0;
  f32 mag = 0.0;

  i32 m = ((W * 3) - 3) - (((W * 3) - 3) & 3);

  for (u64 i = 0; i < (H - 3); i++) {
    for (u64 j = 0; j < m; j+=4) {

      gx += cframe[INDEX(i+0 , j+0 , W * 3)] * -1;
      gx += cframe[INDEX(i+2 , j+0 , W * 3)];
      gx += cframe[INDEX(i+0 , j+1 , W * 3)] * -2;
      gx += cframe[INDEX(i+2 , j+1 , W * 3)] *  2;
      gx += cframe[INDEX(i+0 , j+2 , W * 3)] * -1;
      gx += cframe[INDEX(i+2 , j+2 , W * 3)];

      gy += cframe[INDEX(i+0 , j+0 , W * 3)] * -1;
      gy += cframe[INDEX(i+1 , j+0 , W * 3)] * -2;
      gy += cframe[INDEX(i+2 , j+0 , W * 3)] * -1;
      gy += cframe[INDEX(i+0 , j+2 , W * 3)];
      gy += cframe[INDEX(i+1 , j+2 , W * 3)] *  2;
      gy += cframe[INDEX(i+2 , j+2 , W * 3)];

      mag = sqrt((gx * gx) + (gy * gy));

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

      gx = 0;
      gy = 0;

      [... x4 with j+1 ...]
    }
}
```

0.1 – Implémentation du déroulage x4

J'ai aussi implémenté un déroulage par 8, il sera utile pour la partie vectorisation.

4.2 Vectorisation

La vecotrisation consiste en l'utilisation d'instruction SIMD (Single Instruction Multiple Data), qui permet à une seul instruction de traiter un vecteur entier de données.

4.2.1 Compilateur Auto-Vectorisation

Le compilateur peut lui-même généré du code vectorisé, les flags d'optimisation -O2, -O3 ou -Ofast active cette fonctionnalitée du compilateur. Dans la partie résultats, je montrerai les différents résultats de ces flags sur les codes.

4.2.2 Intrinsics

Ici, je vais utiliser les instructions intrinsics AVX2, les ymm font 256 octets ce qui permet de mettre 8 f32. En s'appuyant sur la version unroll x8 chaque instruction va donc traiter les huits données qui étaient traitées petit à petit dans les huits parties dupliquées de la boucle.

```
void sobel_unroll8_intrinsic_AVX2(f32 *cframe, f32 *oframe, f32 threshold) {

  __m256 v_1 = _mm256_set_ps( 1, 1, 1, 1, 1, 1, 1, 1);
  __m256 v_2 = _mm256_set_ps( 2, 2, 2, 2, 2, 2, 2, 2);
  __m256 v_3 = _mm256_set_ps(-1,-1,-1,-1,-1,-1,-1,-1);
  __m256 v_4 = _mm256_set_ps(-2,-2,-2,-2,-2,-2,-2,-2);

  __m256 threshold_vec = _mm256_set_ps(threshold, ... ,threshold);
  __m256 zero_vec = _mm256_set_ps(0,0,0,0,0,0,0,0);
  __m256 mask_255 = _mm256_set_ps(255,255,255,255,255,255,255,255);

  for (u64 i = 0; i < (H - 3); i++) {
    for (u64 j = 0; j < (W - 3); j+=8) {
      __m256 f1 = _mm256_loadu_ps(&cframe[INDEX(i+0 , j+0 , W)]);
      __m256 f2 = _mm256_loadu_ps(&cframe[INDEX(i+2 , j+0 , W)]);
      __m256 f3 = _mm256_loadu_ps(&cframe[INDEX(i+0 , j+1 , W)]);
      __m256 f4 = _mm256_loadu_ps(&cframe[INDEX(i+2 , j+1 , W)]);
      __m256 f5 = _mm256_loadu_ps(&cframe[INDEX(i+0 , j+2 , W)]);
      __m256 f6 = _mm256_loadu_ps(&cframe[INDEX(i+2 , j+2 , W)]);

      __m256 gx_1 = _mm256_mul_ps(f1 , v_3);
      __m256 gx_2 = _mm256_mul_ps(f2 , v_1);
      __m256 gx_3 = _mm256_mul_ps(f3 , v_4);
      __m256 gx_4 = _mm256_mul_ps(f4 , v_2);
      __m256 gx_5 = _mm256_mul_ps(f5 , v_3);
      __m256 gx_6 = _mm256_mul_ps(f6 , v_1);

      __m256 gx  = _mm256_add_ps(gx_6,_mm256_add_ps(gx_5,
          _mm256_add_ps(gx_4,_mm256_add_ps(gx_3,_mm256_add_ps(gx_1,gx_2)))));

      [... same with gy ...]

      __m256 mag = _mm256_sqrt_ps(
          _mm256_add_ps(_mm256_mul_ps(gx,gx),_mm256_mul_ps(gy,gy)));

      __m256 mask = _mm256_cmp_ps(mag, threshold_vec, _CMP_LT_OS);

      __m256 mask_inverse = _mm256_add_ps(_mm256_mul_ps(mask,v_3),v_1);

      mag = _mm256_add_ps(
          _mm256_mul_ps(mask, mask_255), _mm256_mul_ps(mask_inverse, mag));

      _mm256_store_ps(&oframe[INDEX(i , j , W)], mag);
}}}
```

0.1 – Implémentation avec les intrisics AVX2

4.3 Alignement Mémoire

La version AVX2 utilise des f32, donc sur 4 octets, on veut pouvoir charger 8 valeurs à la fois sur une cacheLine (de 64 octets sur mon PC) sans les couper. Donc on aligne les valeurs sur 32 octets, pour que les valeurs soit bien alignées par rapport à une cacheLine, et que l'on puisse prendre 2 f32 par cacheLine.

4.4 Instructions coûteuses

L'implémentation de base utilise une racine carrée, qui est une instruction coûteuse (12 cycles cpu), pour la supprimée j'ai mis au carré le treshold avec qui elle était comparée. Avec cette approximation, la detection des contours de l'image se dégrade :

FIGURE 1 – Image dégradée sans la racine carrée.

4.5 Parallélisation

Pour paralléliser mon code, j'ai utilisé OpenMP, je ne l'ai testé que sur la version intrinsics du code.

```
#pragma omp parallel
for (u64 i = 0; i < (H - 3); i++) {
    for (u64 j = 0; j < (W - 3); j+=8) {
...
```

0.1 – Boucle parallélisée avec OpenMP

5 Résultats

Dans cette section, je vais présenter les résultats obtenu pour les différentes versions, sur plusieurs compilateurs et avec plusieurs flags d'optimisation.

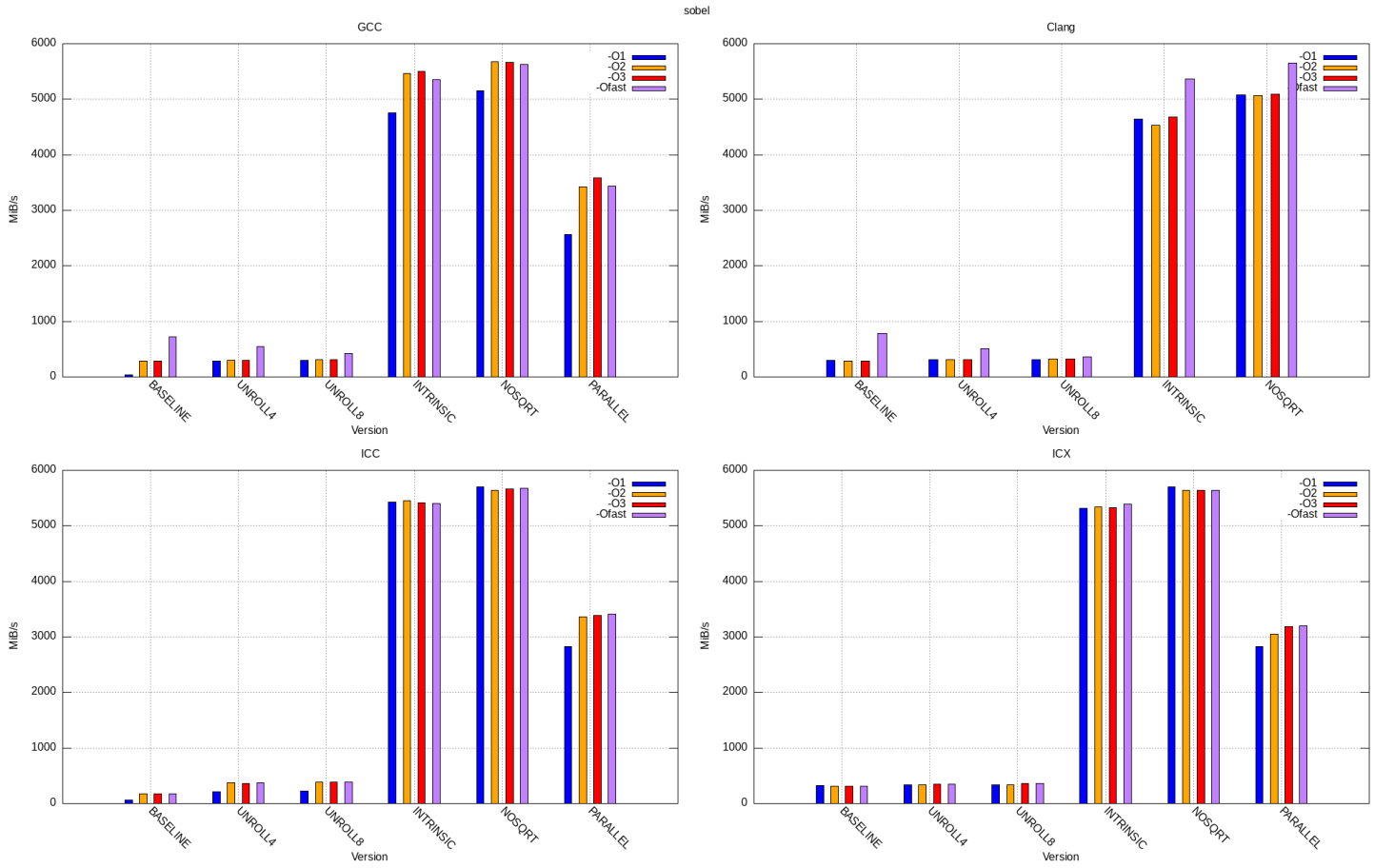

FIGURE 2 – Bande passante en fonction des versions.

Les deux versions d'unroll n'améliorent pas beaucoup la bande passante, c'est surtout la vectorisation qui a un impact. La suppression de la racine carrée augmente encore un petit peu la bande passante. Et au vu des résultats de la version parallèle, je suppose que je l'ai mal implémenté ou que cela ne marche pas bien sur mon PC avec 2 coeurs.

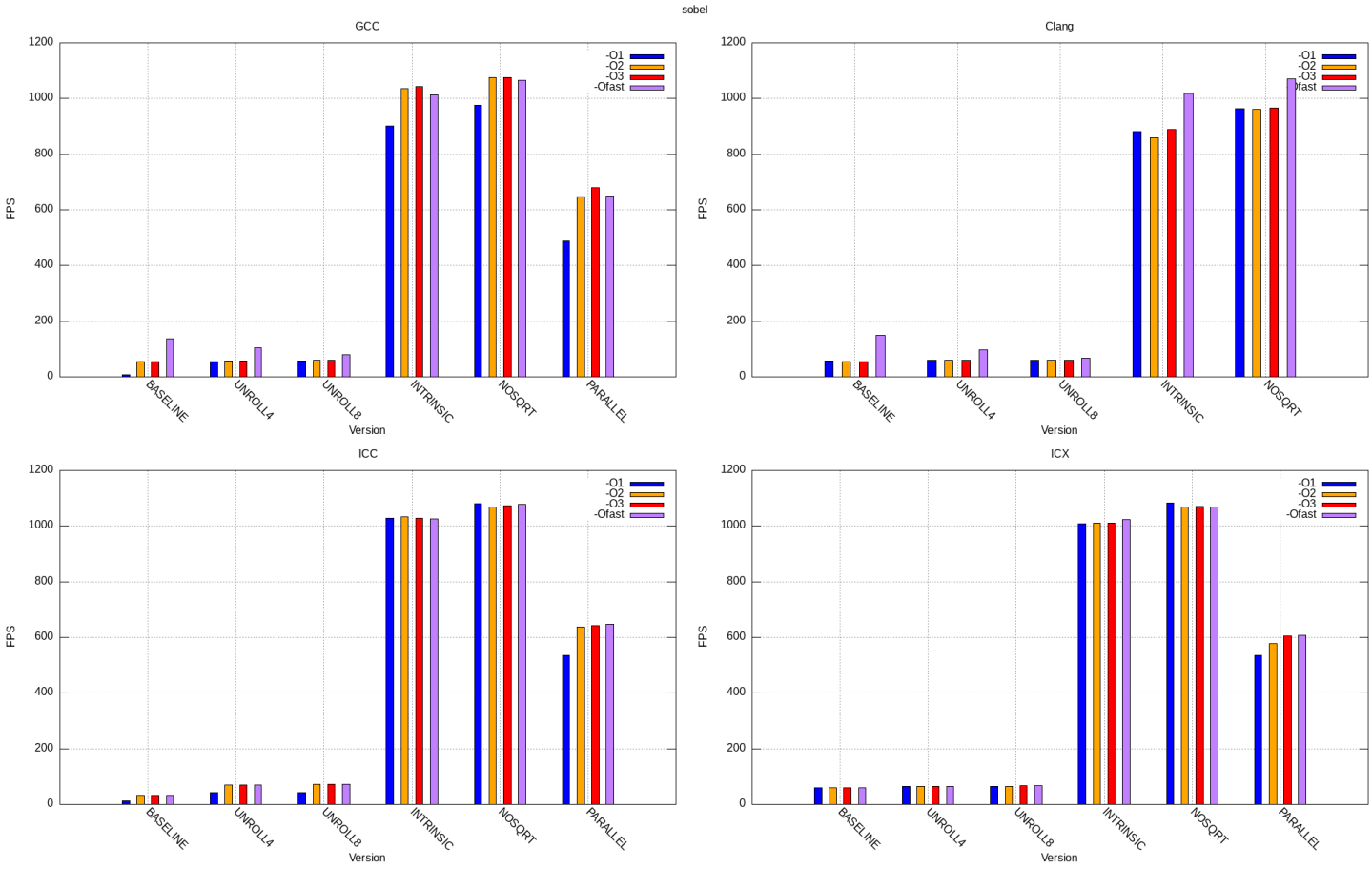

FIGURE 3 – Frame per second en fonction des versions.

J'ai rajouté une deuxième métrique que sont les FPS (Frame Per Second), qui donne le nombre d'images traitées à la seconde (par le kernel du filtre de sobel seulement).

6 Conclusion

Dans ce rapport j'ai parlé des différentes implémentations et optimisations faites pour ce code de détections des contours, déroulage de boucle, vectorisation, parallélisation.

Et j'ai présenté les résultats obtenus sur ma machine avec les compilateurs gcc, clang, icc et icx pour chaque implémentations et avec différents flags d'optimisation.

Pour la suite on pourrait vectoriser avec de l'assembleur directement et utiliser le jeux d'instruction AVX512. On pourrait aussi bufferiser les I/O pour traiter plus d'une image à la fois.