

PAP

# **Projet : rapport3**

4TIN804U

BERASATEGUY Tanguy, GOEDEFROIT Charles

2021-2022

## Table des matières

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>4.5 AVX implementation</b>               | <b>2</b> |
| 1.1      | 4.5.1 The synchronous case . . . . .        | 2        |
| 1.2      | 4.5.2 The asynchronous case . . . . .       | 2        |
| <b>2</b> | <b>4.7 OpenCL Implementation</b>            | <b>4</b> |
| 2.1      | 4.7.1 Basic OpenCL Implementation . . . . . | 4        |

# 1 4.5 AVX implementation

## 1.1 4.5.1 The synchronous case

On fait le speedup avec *omp\_tile* entre les tiling *opt* et *avx* sur la machine *Uhura* on obtient 4483 pour *opt* et 4764 pour *avx* un speedup de  $1.15 = \frac{15236.266}{13207.626}$  sur une exécution avec *tw*=64, *th*=512, 1 thread et une taille de 512. Il n'y a pas une grande différence entre les 2 version car gcc a une très bonne vectorization.

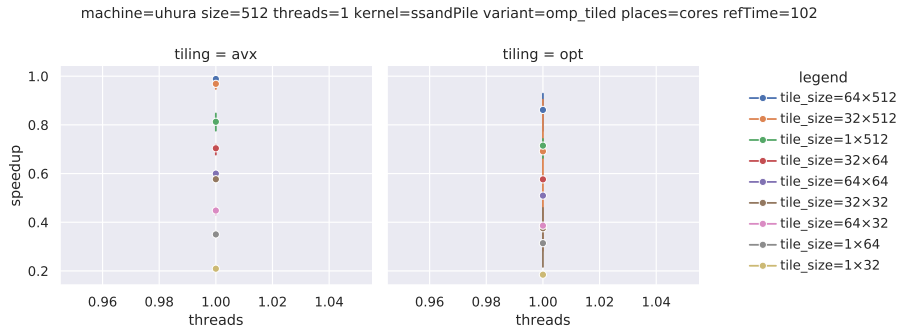


FIGURE 1 – ssandPile comparaison avx et opt sur certaines tailles de tuiles

On voit que le speed-up de l'avx est supérieur à la version opt, et avec des marges d'erreurs moins conséquentes. (Ayant du mal avec les fichiers d'expériences, on ne sait pas sur quoi il se base pour les speed-up)

On fait le speedup avec *omp\_lazy* entre les tiling *opt* et *avx* sur la machine *troi* on obtient 4483 pour *opt* et 4764 pour *avx* un speedup de  $1.24 = \frac{6464.618}{5189.285}$  sur une exécution avec *tw*=64, *th*=512, 1 thread et une taille de 512. Il n'y a pas une grande différence entre les 2 version car gcc a une très bonne vectorization.

## 1.2 4.5.2 The asynchronous case

On a implémenter en suivant la consigne sur sujet. La version *avx* fonction avec la variante *omp\_tiled* avec un speedup de  $1.01 = \frac{9455.038}{9349.466}$  : Ici encore, nous ne sommes pas très loin de la version *opt* car le compilateur vectorise bien.

Par-contre la version *avx* ne fonctionne pas avec la variante *omp\_lazy*.

Le code de la fonction :

```
1 int asandPile_do_tile_avx(int x, int y, int width, int height)
2 {
3     if (x == (DIM - 1) - width)
4         x -= 1;
5
6     //  $\overrightarrow{X} == \text{vecX}$ 
7     const __m256i vec3_i = _mm256_set1_epi32(3);
8     const __m256i vec0_i = _mm256_set1_epi32(0);
9
10    int diff = 0;
11    for (int j = y; j < y + height; j++)
```

```

12     for (int i = x; i < x + width; i += AVX_VEC_SIZE_INT)
13     {
14         // vecT_{j-1,i} <-- (t_{j-1,i+k}, ..., t_{j-1,i})
15         __m256i topVec_i = _mm256_loadu_si256((__m256i *) &table(in, j - 1, i));
16         // vecT_{j,i} <-- (t_{j,i+k}, ..., t_{j,i})
17         __m256i vec_i = _mm256_loadu_si256((__m256i *) &table(in, j, i)); // load?
18         // vecT_{j+1,i} <-- (t_{j+1,i+k}, ..., t_{j+1,i})
19         __m256i bottomVec_i = _mm256_loadu_si256((__m256i *) &table(in, j + 1, i));
20
21         // vecD <-- vec_i / 4
22         __m256i vecD = _mm256_srli_epi32(vec_i, 2);
23
24         // (vecD << 1)
25         __m256i vecDShiftLeft = _mm256_alignr_epi32(vecD, 1);
26
27         // (vecD >> 1)
28         __m256i vecDShiftRight = _mm256_alignr_epi32(vecD, 7);
29
30         // vec_i <-- vec_i % 4 + vecDShiftLeft + vecDShiftRight
31         __m256i res_vec_i = _mm256_add_epi32(_mm256_and_si256(vec_i, vec3_i),
32                                             _mm256_add_epi32(vecDShiftLeft, vecDShiftRight));
33
34         // topVec_i <-- topVec_i + vecD
35         topVec_i = _mm256_add_epi32(topVec_i, vecD);
36
37         // bottomVec_i <-- bottomVec_i + vecD
38         bottomVec_i = _mm256_add_epi32(bottomVec_i, vecD);
39
40         // t_{j,i-1} <-- t_{j,i-1} + vecD[0]
41         __m256i leftVec_i = _mm256_loadu_si256((__m256i *) &table(in, j, i - 1));
42
43         leftVec_i = _mm256_add_epi32(leftVec_i, vecD);
44
45         _mm256_storeu_si256((__m256i *) &table(out, j, i - 1), leftVec_i);
46
47         // t_{j,i+k+1} <-- t_{j,i+k+1} + vecD[k] : k = AVX_VEC_SIZE_INT-1
48         __m256i rightVec_i = _mm256_loadu_si256((__m256i *) &table(in, j, i + 1));
49
50         rightVec_i = _mm256_add_epi32(rightVec_i, vecD);
51
52         _mm256_storeu_si256((__m256i *) &table(out, j, i + 1), rightVec_i);
53
54         // (t_{j-1,i+k}, ..., t_{j-1,i}) <-- vecT_{j-1,i}
55         _mm256_storeu_si256((__m256i *) &table(out, j - 1, i), topVec_i);
56         // (t_{j,i+k}, ..., t_{j,i}) <-- vecT_{j,i}
57         _mm256_storeu_si256((__m256i *) &table(out, j, i), res_vec_i);
58         // (t_{j+1,i+k}, ..., t_{j+1,i}) <-- vecT_{j+1,i}
59         _mm256_storeu_si256((__m256i *) &table(out, j + 1, i), bottomVec_i);
60
61
62         __m256i mask = _mm256_xor_si256(res_vec_i, vec_i);
63         if (_mm256_testz_si256(mask, mask) != 1)

```

```
64         diff = 1;
65     }
66
67     return diff;
68 }
```

---

## 2 4.7 OpenCL Implementation

### 2.1 4.7.1 Basic OpenCL Implementation

sur la machine *troi* on a executer la version ocl avec une taille de 1024x1024 et des tuiles de taille 16x16, les 69191 iterations sont executer en 1648ms.

sur la machine *troi* on a executer la version omp\_tiled avec une taille de 1024x1024 et des tuiles de taille 16x16 et le tuilage opt, cela ce fini au bout de ...ms en ... iterations.

le speedup et de  $4.81 = \frac{332871}{69191}$ .