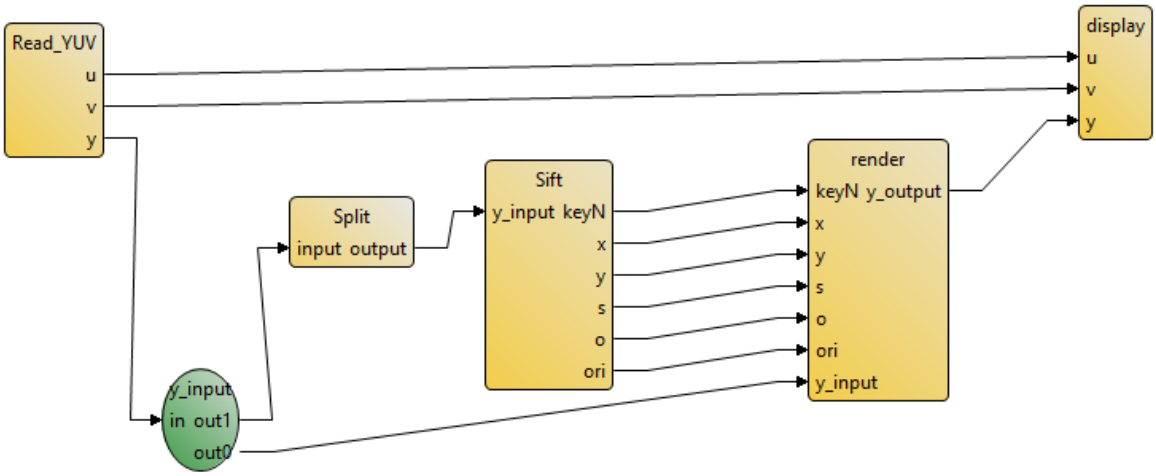
Parallélisation d'un algorithme SIFT

Nous avons choisi d'implémenter un algorithme SIFT (Scale-invariant feature transform) qui permet d'extraire des descripteurs SIFT de l'image analysée. Ceux-ci sont intéressants en vision car peu dépendants de l'échelle, du cadrage, de l'angle d'observation et de l'exposition de l'image.

Nous avons récupéré un code sur Internet qui fonctionnait correctement (voir https://sites.google.com/site/5kk73gpu2011/assignments/sift).

Première approche : découpage de l'image en vue d'une parallélisation

Nous avons d'abord pensé à découper l'image afin de traiter N tranches sur N cœurs.



Le bloc Split découpait sommairement l'image en N tranches (sans recouvrement) et le bloc render récupérait les données des descripteurs (position, orientation et intensité). Les vecteurs sont affichés sous forme de cercles (pour une meilleure visibilité de leur norme) alors qu'un trait indique leur direction. Par ailleurs, les pixels sont blancs sur fond sombre et noirs sur fond clair.

Cette solution n'était malheureusement pas efficace puisqu'en découpant l'image sans recouvrement, le résultat était biaisé. Il aurait fallu prévoir un recouvrement. Nous avons ensuite préféré paralléliser les calculs plutôt que d'évaluer le recouvrement nécessaire au bon fonctionnement de cette version.

Seconde approche : parallélisation des calculs

Optimisation de la mémoire

Le code que l'on a utilisé contenait plusieurs malloc qui pouvaient être très problématiques. Nous avons éliminé assez simplement la plupart d'entre eux en estimant les valeurs maximales considérées pour l'allocation dynamique. Cependant, les malloc concernant les octaves, les « scale spaces » et les « difference of gaussians » ont posé plus de problème car dépendant de la résolution initiale. Nous avons optimisé l'utilisation de la mémoire sachant que chaque octave voit sa taille divisée par 4 (hauteur et largeur divisées par 2), ce qui fait que la taille des octaves tend vers 0. Pour l'ensemble de celles-ci, il suffit du double de la résolution d'image initiale. Preesm alloue cette mémoire globale et il suffit ensuite de jouer sur les adresses des octaves pour le faire correspondre. Nous faisons de même pour les « scale spaces » et les « differrence of gaussians », qui nécessitent un peu plus de réflexion car se situant sur des tableaux de float***.

```
void buildSS(unsigned char* octaves_in,
             float* scale_space_in,
             int *O, int S,
             int* octavesW, int* octavesH,
             float* sigmas){

    int i, j;
    unsigned char* octaves[MAX_O];
    float* scaleSpace[MAX_O][MAX_S];
    int posO=0,posSS=0, octaveSize=0;

    octaves[0]=octaves_in;
    for(i = 1; i < *O; i++){
        posO+=octavesH[i-1]*octavesW[i-1];
        octaves[i]=octaves_in+posO*sizeof(unsigned char);
    }
    for(i = 0; i < *O; i++){
        octaveSize=octavesH[i]*octavesW[i];//*sizeof(float);
        for(j = 0; j < S; j++){
            scaleSpace[i][j] =scale_space_in + posSS;
            posSS+=octaveSize;
        }
    }
```

Ci-dessus, octaves_in et space_scale_in correspondent aux tableaux alloués par Preesm. Octaves et scaleSpace sont des tableaux de 2 et 3 dimensions qui prennent en compte de la taille de l'image des images pour le traitement, à partir des adresses fournies par Preesm. C'est une manière peu propre pour gérer la mémoire mais elle permet de l'optimiser et surtout d'éviter les allocations dynamiques. Sur PC, nous avons obtenu un gain de plus de 30% de cette manière, sans paralléliser l'algorithme.

Blocs à paralléliser

Par ailleurs, nous avons évalué le temps nécessaire au calcul de chacun des sous-blocs de l'algorithme afin de déterminer lequel était le plus intéressant à paralléliser.

init time: 0.021411 ms
buildOB time: 0.090297 ms
buildSS time: 50.590237 ms
DoG time: 2.060696 ms
extreme time: 6.115685 ms
orientation time: 0.575605 ms

Total time: 60.286461 ms

On voit clairement que le bloc qui nous intéresse le plus est ici builds, qui représente plus de 80% du temps de calcul d'une image.

Parallélisation du code

C'est donc le bloc buildSS que nous avons décidé de paralléliser. Les principales boucles for parcourent les octaves et les scale spaces afin de calculer leur gaussienne. Nous avons décidé de découper ce bloc en 8 afin de paralléliser ses calculs. A partir de là, il suffisait de récupérer les résultats qui nous intéressaient depuis chacun de ses sous-blocs afin de continuer le traitement. Malheureusement, ce code ne fonctionne que pour les résolutions permettant d'avoir 6 octaves et 5 scale spaces. Chacun des 8 blocs peut être exécuté indépendamment. Ainsi, en configuration mococoeur, ils sont exécutés séquentiellement alors que deux blocs par cœur sont exécutés pour 4 cœurs par exemple. Sur 8 cœurs, chacun d'entre eux exécute un mini-bloc. Sur PC, le code est correctement schédulé et exécuté sur 1, 2 ou 4 cœurs.

Conclusion

Nous n'avons pas pu implémenter l'algorithme sur la carte avec 8 cœurs en raison d'un problème de stack trop grande (stack à augmenter également sur CodeBlocks). Cependant, il suffit d'effectuer quelques ajustements de mémoire (mémoire allouée par Preesm plutôt que sur la stack). L'algorithme est autrement entièrement fonctionnel et parallélisable jusqu'à 8 cœurs.