



DETECTION DE CONTOUR SUR CARTE DSP

Rapport

PAR

**ZENGZHE WANG
CHARLES PAULAS VICTOR**

ENCADRE PAR HABIB MEHREZ

MAI 2017

1. Introduction	2
2. Approche	2
3. Environnement de développement	4
4. Développement:	5
4.1. Structuration du C	5
4.2. Assembleur	5
4.3. Assembleur optimisé	6
5. Résultats:	7
5.1. Résultats code C	7
5.2. Résultats C avec assembleur	8
5.3. Résultats C avec assembleur optimisé	8
5.4. Comparaison	9
6. Conclusion	9
7. Bibliographie	9

1. Introduction

Dans le cadre de l'UE ANUMDSP, un projet traitement d'image sur une carte DSP nous a été proposé.

Le traitement d'image est un domaine interdisciplinaire en pleine croissance et nécessite de nombreux outils issues de l'Électronique, de l'Informatique et des Mathématiques. L'un des nombreux défis de ce domaine est la détection d'objets. En effet, cette fonctionnalité s'avère très utile dans de nombreux secteur :

- Santé : détection des cellules cancéreuses dans une image IRM
- Voiture autonome : analyse et décision à partir de caméras
- Industrie : détection d'usures sur des machines, mesures de la qualité des soudures sur les circuits intégrés
- Physique : traitement d'images satellite

Un des éléments importants dans la détection d'objet est la capacité de décrire le contour de l'objet. Ainsi nous réaliserons la détection de contour par convolution.

Par ailleurs, nous choisissons d'effectuer le traitement de façon numérique. En effet, le numérique nous épargne beaucoup de désagréments en comparaison à un traitement analogique, tels que l'influence du bruit, la consommation énergétique et permet d'avoir plus de polyvalence.

Néanmoins, les calculs ne seront pas effectués par un ordinateur mais une carte DSP, capable de s'intégrer dans un système embarqué. En effet, les cartes DSP sont des processeurs dont le matériel, le logiciels et le jeu d'instruction sont optimisés pour un traitement haute performance de signaux numériques. Par exemple, en comparaison d'un CPU classique, un DSP est capable d'avoir des performances similaires mais pour une fréquence d'horloge inférieure, ce qui permet d'économiser en consommation d'énergie.

De plus, les cartes DSP disposent d'architectures proposant de possibles améliorations que nous mettrons en place.

2. Approche

Pour effectuer la détection de contour, nous appliquons une convolution à l'image source.

Dans ce projet, nous travaillerons avec des images 2D en nuance de gris.

La convolution peut se modéliser mathématiquement par cette formule :

$$Y(i,j) = \sum_{k=0}^{N} \sum_{l=0}^{M} h(k,l) \times I(i-k, j-l)$$

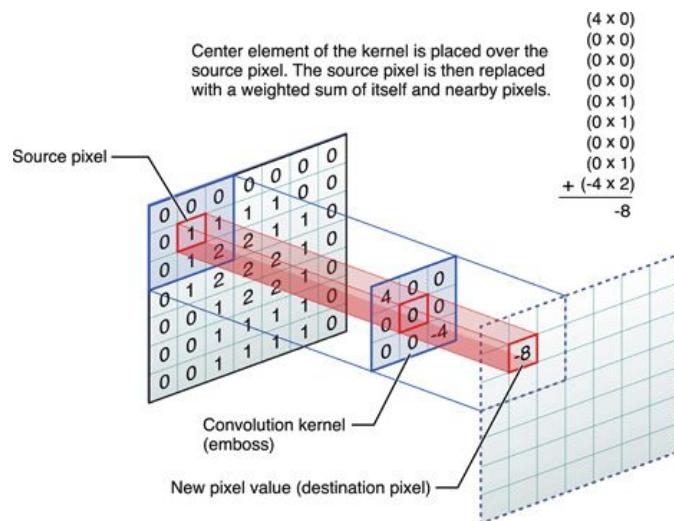
I est la matrice de l'image source ayant N pixels.

H est la matrice masque ayant M pixels. Le masque est l'élément déterminant la nature du filtre. Ainsi les valeurs de cette matrice détermine le mode de fonctionnement du filtre.

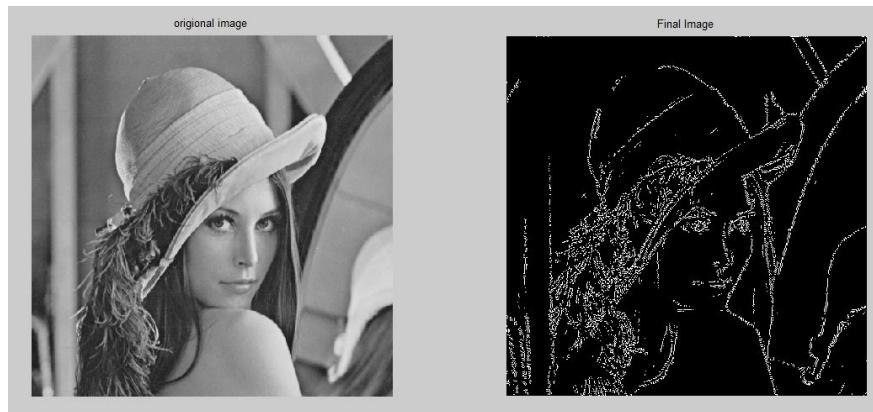
-1/8	-1/8	-1/8
-1/8	1	-1/8
-1/8	-1/8	-1/8

Matrice masque pour la détection d'image

La convolution s'applique en faisant parcourir ce masque sur chaque pixel de l'image source. L'application consiste à multiplier toutes les valeurs du masque à une portion de pixel (provenant de l'image source) de même gabarit que le masque et de remplacer dans l'image résultat, à la position correspondante au centre de la portion de pixel, par la valeur de la somme de ces multiplications.



Exemple de traitement pour un pixel



Exemple de détection de contours

3. Environnement de développement

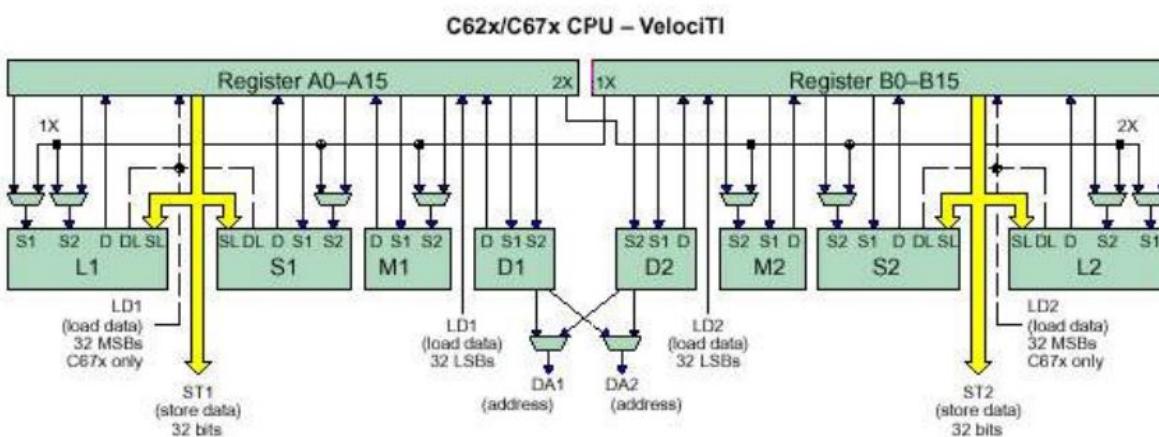
Tout d'abord le format d'image sur lequel nous travaillons est le portable graymap file format (.pgm) et plus particulièrement dans la version ASCII.

L'intérêt de ce format est la possibilité de connaître la valeur du pixel juste en lisant dans un fichier. Il n'y pas une procédure de décodage spécifique.

La première ligne indique la version du pgm (ASCII :P1 ou binaire : P5). La seconde ligne indique la longueur et largeur de l'image.

```
P1
# Un exemple bitmap de la lettre "J"
7 10
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 1 0 0 0 1 0
0 0 1 1 1 0 0
0 0 0 0 0 0 0
```

Dans ce projet, il nous est proposé de travailler avec la carte DSP de chez Texas Instrument TMS 320 C6713, cadencé à 25 MHz, 1800 MIPS et 1350 MFLOPS



Son avantage réside dans une architecture orientée au traitements du signal avec notamment des sous unité de calculs dédiés pouvant fonctionner en parallèle. Cette spécificité sera exploitée dans notre projet grâce à des appel d'une fonction écrite en assembleur.

Pour travailler avec la carte, Texas Instrument fournit un outil appelé Code Composer Studio (CSS). C'est un logiciel qui combine à la fois un éditeur de code, un

compilateur, un débogueur ainsi que des outils d'analyse de performance. Par ailleurs, il prend en charge la gestion de la communication entre le PC et la carte (chargement des exécutables sur la carte par exemple).

4. Développement:

4.1. Structuration du C

Le code C est notre version de départ. Elle est composé des fonctions qui nous permettront d'ouvrir, écrire et fermer de le fichier PGM. En effet, nous prenons en entrée l'image source et on la stock dans la mémoire de la carte. Lors de traitement, les pixels résultats sont stockés dans un autre espace mémoire. Cet espace mémoire est ensuite utilisé pour réaliser l'image résultat (toujours pgm ascii).

Pour effectuer le traitement, nous mettons en place un certain formalisme. Nous stockons dans deux tableaux (de a taille du masque, ici neuf) une portion de l'image source (pix_buf[9]) et les coefficients du masque (coeff_buf[9]). Ainsi le tableau contenant la portion d'image source sera mis à jour à chaque pixel traité. Un traitement consiste à multiplier les deux tableaux entre eux élément par élément et de faire une somme des résultats des multiplications :

```
for i : 0 -> 9
    res += pix_buf[i] * coeff_buf[i]
end
```

Ce traitement sera assigné une fonction en c qui sera appelé dans le main(). Nous constatons que les pixels du bord de l'image posent problème. Il sera corrigé par un traitement particulier du bord : les pixels manquant seront remplacés par des copies des pixels présents.

D'autre part, nous avons présenté des coefficients pour le masque égale à $\frac{1}{8}$. Or ceci entraînera un grand nombre de divisions (opération très gourmand en cycle), ainsi la division par 8,0 sera faite sur le résultat finale afin d'économiser des cycles.

4.2. Assembleur

Nous constatons que la tâche la plus gourmande est la séquence de multiplication. Nous décidons donc de la faire réaliser à l'aide d'un code assembleur.

```
.global _fct_asm
_fct_asm:
    ZERO    .L1    A9
    MVK     .S1    9,A2
LOOP:  LDBU   .D1    *A4++,A5
    NOP     4
    LDB     .D3    *B4++,B5
```

```

NOP      4
MPYUS   .M1X    A5,B5,A6
NOP
ADD     .L1     A9,A6,A9
[A2]   SUB     .L1     A2,1,A2
[A2]   B       .S1     LOOP
NOP      5
MV      .L1     A9,A4
B       .S2     B3
NOP      5

```

Il s'avère qu'il est possible d'optimiser ce code grâce à différente technique pour gagner en performance.

4.3. Assembleur optimisé

En effet, en déroulant la boucle, nous pouvons enlever la soustraction pour la boucle mais surtout solliciter quasi toutes les unités de calculs de la carte DSP. Ainsi les performances sont fortement améliorées.

Pour l'usage du parallélisme, il ne faut pas oublier d'utiliser le symbole || avant l'instruction.

```

.global _fct_asm_opt
_fct_asm_opt:
    ZERO   .L1    A9

    LDBU   .D1    *A4++,A5
||  LDB    .D3    *B4++,B5

    LDBU   .D1    *A4++,A5
||  LDB    .D3    *B4++,B5
||  MPYUS .M1X    A5,B5,A6

    LDBU   .D1    *A4++,A5
||  LDB    .D3    *B4++,B5
||  MPYUS .M1X    A5,B5,A6

    LDBU   .D1    *A4++,A5
||  LDB    .D3    *B4++,B5
||  MPYUS .M1X    A5,B5,A6

```

```

|| ADD    .L1    A9,A6,A9

|| LDBU   .D1    *A4++,A5
|| LDB    .D3    *B4++,B5
|| MPYUS  .M1X   A5,B5,A6
|| ADD    .L1    A9,A6,A9

|| ADD    .L1    A9,A6,A9

|| ADD    .L1    A9,A6,A9

MV     .L1    A9,A4
B      .S2    B3
NOP    5

```

5. Résultats:

5.1. Résultats code C



Résultat 128 x 128

	Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
	00000000-00000064	fct_j	288-296-fct_p.c	function	4096	9290730	9290730

Analyse de profiler

On obtient 9290730 cycles.

5.2. Résultats C avec assembleur



Résultat 128 x 128

Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0x0ec00-0x0ec9c	rct_sen.sen(3:17\$)	3-17:rct_sen.sen	function	4096	3940639	3940639
+0						
+1						
+2						

Analyse de profiler

On obtient 3940639 cycles.

5.3. Résultats C avec assembleur optimisé



Résultat 128 x 128

Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0x0ec00-0x0ec9c	rct_sen_opt.sen(3:17\$)	3-17:rct_sen_opt.sen	function	4096	2805654	2805654
+0						
+1						
+2						

Analyse de profiler

On obtient 2805654 cycles.

5.4. Comparaison

Version	Cycles	Différence avec le C	Rapport (%)
code C	9290730	0	0
code C + asm	3940639	5350091	42,41474028
code C + asm optimisé	2805654	6485076	30,19842359

Analyse de profiler

Nous constatons que l'utilisation simple de d'instruction assembleur permet de diviser par deux le nombre de cycle.

Par ailleurs, en optimisant le code assembleur et par conséquent en profitant des possibilités de parallélisme offert par la carte DSP, nous obtenons un peu moins d'un tiers du nombre de cycle de la version C (30,19 %).

6. Conclusion

Nous comprenons donc que l'usage de carte DSP est surtout légitime lorsqu'on associe le code C et l'assembleur.

Il s'avère qu'il est d'habitude de prototyper le code C sur un PC et ensuite de l'implémenter sur carte DSP. Il faut donc après avoir prouvé le bon fonctionnement, essayer d'isoler les tâches gourmande en nombre de cycle et de traduire ces tâches en code assembleur.

7. Bibliographie

- <http://www.analog.com/en/analog-dialogue/articles/dsp-101-part-1.html>
- <https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html>