Université de Paris Sud

Master IST - M2Pro

DSP

Filtrage médian

1 Introduction

Le filtrage médian est un filtre non linéaire de la famille des filtres d'ordre. Le filtrage médian est très robuste à différents types de bruit, comme le bruit gaussien ou le bruit impulsionnel. Le principal inconvénient du médian est sa complexité mathématique en $O(n^2)$ pour calculer le médian de n éléments. Son autre inconvénient est que dans sa formulation classique (sous forme de tri) il est impossible de paralléliser certaines parties de l'algorithme à cause des dépendances de données. Le but de ce TP est de réaliser différentes implantations du filtre médian notamment l'implantation d'un algorithme parallélisable par morceau, qui pourra tirer partie des nombres unités de calcul d'un processeur VLIW.
L'ensemble du TP est réalisable en C sur une machine classique, seule la dernière partie (assembleur linéaire) ne peut se faire que sous l'environnement *Code Composer Studio*.
Un type utilisateur (*user data type*) va être défini pour séparer les variables de l'algorithme des variables liées au codage des boucles : `typedef short int16 ;` qui devra être présent dans votre code C.

2 Médian classique

L'algorithme classique du filtre médian repose sur l'utilisation d'un algorithme de tri (tri par sélection, tri par insertion, tri à bulles, tri rapide). Ces tris ont une compléxité en $O(n^2)$ sauf le dernier qui est en $n \log(n)$, car il est nécessaire de réaliser n^2 (respectivement $n \log(n)$) comparaisons pour trier n valeurs. Une fois ces n valeurs triées, la valeur médiane est la valeur se trouvant au milieu, en $n/2$. Dans toute la suite, n sera égal à 9.
L'implantation classique se fera en suivant les étapes suivantes :

1. Ecrire une fonction `median9Array` qui utilise l'algorithme de tri par sélection et qui renvoie la valeur médiane. Le prototype de la fonction est :
   `int16 median9(int16 a, int16 b, int16 c, int16 d, int16 e, int16 f, int16 g, int16 h, int16 i) ;`
2. Valider l'algorithme (en mode *Debug*) avec comme valeurs de test les valeurs suivantes : `9,7,8,3,1,2,6,4,5`.
3. Mesurer les performances de votre implantation (en mode *Release*) via le filtrage d'un tableau de de $N = 1000$ valeurs générées aléatoirement via la fonction `void rand16(short *T, int n) ;`. Le nom du tableau source sera `X` et le nom du tableau destination sera `Y`. Ces tableaux pourront être défini de manière statique `int16 X[N], Y[N]`.

La fonction `rand16` utilisera un générateur aléatoire à congruence linéaire :

$$y = f(x) = (ax + b) \bmod c \qquad (1)$$

avec a, b, c premiers entre eux (on pourra prendre $a = 11$, $b = 13$, $c = 17$).

3 Médian rapide

3.1 Présentation de l'algorithme

Lorsque n est un carré, il existe un algorithme bien plus astucieux et qui peut être paralléliser par partie. L'algorithme est le suivant (dans notre cas, nous avons 3×3 valeur à trier) :
Soient les 9 valeurs suivantes à trier

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{2}$$

- étape 1 : trier chaque ligne, tri de $\{a, b, c\}$ tri de $\{d, e, f\}$ et tri de $\{g, h, i\}$
- étape 2 : trier chaque colonne, tri de $\{a, d, g\}$, tri de $\{b, e, h\}$, tri de $\{c, f, i\}$
- étape 3 : trier la diagonale secondaire, $\{g, e, c\}$: le médian se trouve alors au milieu e

3.2 Démonstration

Chaque étape crée une relation d'ordre partiel par ligne (respectivement par colonne). La figure 1 décrit la construction de ces relations d'ordre. La flèche signifiant la relation "*supérieur à*"

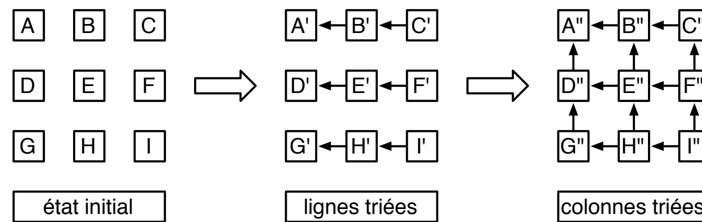

Fig. 1 – tri par ligne et par colonne

En appliquant une une rotation de 45 degré au schéma, (figure 2), on voit que les trois points du haut (A", D" et B") sont les trois plus petits (même si D" et B" ne sont pas triés). De même, les trois points du bas (H", F" et I") sont les trois plus grands. La médiane est donc forcément parmi les trois points du milieu (G", E" et C") qu'il ne reste plus qu'à trier. Une ces trois points triés, la valeur est en E. CQFD !

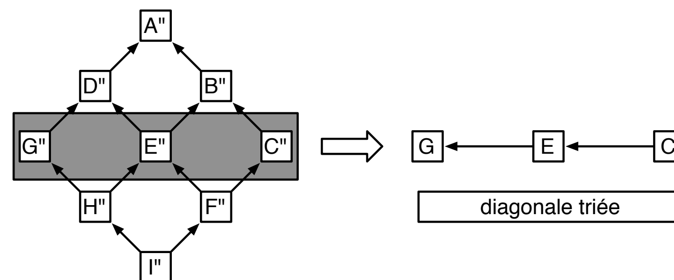

Fig. 2 – Extraction du médian

3.3 Implantation du tri

Au lieu d'utiliser le tri par sélection ou le tri par insertion pour trier les valeurs par 3 (en ligne ou en colonne), c'est le tri à bulles qui va être utilisé car il a la particularité d'avoir une structure itérative régulière. Il s'implante particulièrement bien en VHDL. Soit l'opérateur mM (pour minMax) qui prend en entrée deux valeurs et qui renvoie les deux valeurs triées (la valeur de gauche est la valeur min, la valeur de droite est la valeur max) :

$$mM(a, b) = (\min(a, b), \max(a, b)) \tag{3}$$

L'opérateur mM peut aussi être vu comme une simple permutation si $b < a$ (dans le cas contraire rien n'est fait) :

$$mM(a, b) = \begin{cases} (b, a) & \text{si} \quad \text{b<a} \\ (b, a) & \text{sinon} \end{cases} \tag{4}$$

Trier a, b, c consister à :

1. trier a, b : $mM(a, b)$
2. trier b, c : $mM(b, c)$
3. trier a, b : $mM(a, b)$

A la fin la plus petite valeur est en a, la plus grande est en c et la valeur médiane au milieu. C'est le principe du tri à bulles qui fait remonter les plus petites valeurs en début de tableau, et les plus grandes en fin de tableau (figure 3).

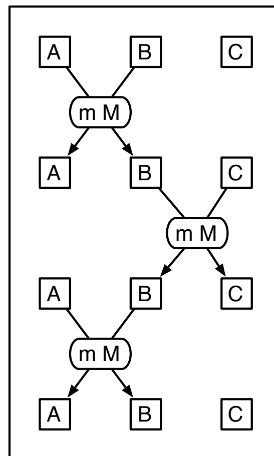

FIG. 3 – principe du tri à bulles appliqué à 3 valeurs

3.4 Implantation via des fonctions

1. Faire un schéma montrant les différents calculs pouvant être fait en parallèle
2. Ecrire une fonction qui renvoie le minimum de 2 nombres, dont le prototype est :
   `int16 min16(int16 a, int16 b) ;`
3. Ecrire une fonction qui renvoie le maximum de 2 nombres, dont le prototype est :
   `int16 max16(int16 a, int16 b) ;`

4. Ecrire une fonction `minMax16` utilisant les fonctions `min16` et `max16` avec passage par adresse des arguments et qui renvoie dans le premier argument le minimum et dans le second le maximum. Le protype de la fonction est :
   `void minMax16(int16 *a, int16 *b) ;`
5. Ecrire une fonction `tri` qui tri valeurs valeurs passées par adresse. Cette fonction utilisera la fonction `minMax16`. Le prototype de la fonction est :
   `void tri(int16 *a, int16 *b, int16 *c) ;`
6. Ecrire une fonction `median9` qui prend 9 valeurs en argument et qui implante l'algorithme rapide. Le prototype de la fonction est :
   `int16 median9(int16 x0, int16 x1, int16 x2, int16 x3, int16 x4, int16 x5, int16 x6, int16 x7, int16 x8) ;`
7. Valider l'algorithme (en mode *Debug*)
8. Mesurer les performances (en mode *Release*). Quel est le gain ?
9. La fonction `minMax16` étant appelée très souvent, proposer une autre implantation, avec toujours le même prototype

3.5 Implantation via des MACROS

Afin d'accélérer cet algorithme, on décide de remplacer les appels de fonctions par des appels de macros (une solution intermédiaire aurait été d'appliquer des fonctions d'*inlining*).

1. Ecrire une macro `MINMAX16(a,b)`
2. Ecrire une macro `TRI(a,b,c)`
3. Ecrire une macro `MEDIAN9(a,b,c,d,e,f,g,h,i)`
4. Valider l'algorithme (en mode *Debug*)
5. Mesurer les performances (en mode *Release*) de cette nouvelle implantation. Quel est le gain ?

3.6 Implantation via l'assembleur linéaire

Le langage C est très riche mais ne permet pas d'exprimer certaines optimisations liées à l'architecture du DSP, comme :

- les instructions exécutées conditionnellement à l'état d'une variable : `[cond] ADD a,b,c`
- les instructions pouvant s'exécuter en parallèle, au même cycle : `ADD a,b,c || ADD e,f,g`
- toute combinaison d'instructions parallèles à exécution conditionnelle.

1. Ecrire une macro assembleur `MINMAX16(a,b)`
2. Ecrire une macro assembleur`TRI(a,b,c)`
3. Ecrire une fonction assembleur`MEDIAN9_SA` qui renvoie le médian. Le prototype de la fonction est :
   `MEDIAN9_SA(a,b,c,d,e,f,g,h,i)`
4. Valider l'algorithme (en mode *Debug*)
5. Mesurer les performances (en mode *Release*) de cette nouvelle implantation. Quel est le gain ?