

filtrage

Compte rendu de TP à rendre pour la semaine suivante dernier délai

1 Produit scalaire

Codez en C, différentes implantations du produit scalaire. Les données utilisées seront globales :

```
typedef byte int8;
typedef short int16;
typedef int int32;
#define N 1000
int16 X[N];
int16 Y[N];
```

Les prototypes des fonctions seront du type :

```
int32 dotProduct(int16 *pX, int16 *pY, int n)
```

Les appels se feront par passage d'arguments :

```
dotProduct(X, Y, N);
```

1.1 Partie théorique

Pour chaque version faire un chronogramme du code, calculez la complexité en terme d'entrées/sortie (*IO*) de multiplications (*MUL*) d'additions/soustractions (*ALU*). Indiquez le nombre de cycle d'une boucle et le *cpp*. Chaque fonction mettra en oeuvre différentes optimisations :

- version Basique (`dotProduct_B(int16 *pX, int16 *pY, int n)`) : C classique.
- version ILU2 (`dotProductILU2(int16 *pX, int16 *pY, int n)`) : Loop Unrolling Interne d'ordre 2. A quoi faut il faire attention ? Faire un schéma.
- version MPY (`dotProduct_MPY(int16 *pX, int16 *pY, int n)`) : utilisation des *intrinsics* `_mpy` (low x low) `_mpyh` (high x high) `_mpylh` (low x high) et `_mphi` (high x low) : les pointeurs sur tableaux sont castés en `int` afin de gérer des chargement 32 bits (donc de 2 int16 16-bits). Combien y a t il d'itération à faire dans la boucle ?
- version DOPT (`dotProduct_DOTP(int16 *pX, int16 *pY, int n)`) : utilisation de l'*intrinsic* `_dotp2`, les pointeurs sur tableaux sont castés en `int` afin de gérer des chargement 32 bits (donc de 2 short 16-bits). Combien y a t il d'itération à faire dans la boucle ?

1.2 Partie pratique

Pour chaque version, analysez le code asm généré : indiquez la complexité pratique, le nombre de cycle pour une boucle et le *cpp*. (Attention, certaines instructions de multiplication (comme `mpy2` ou `dotp2`) réalisent deux opérations de multiplication : faites attention dans votre décompte). Comparez résultats théoriques et pratiques.

1.3 Assembleur linéaire

Voici le code (1) du produit scalaire en assembleur linéaire.

- Les directives `.sect` `.global` sont nécessaires pour l'édition des liens. Le code C appelant cette fonction doit indiquer que cette fonction est externe : `extern int dotProduct_SA(int16 *pX, int16 *pY, int n);`

```

.sect ".text :_dotProduct_SA"
.global _dotProduct_SA
_dotProduct_SA .cproc px,py,n
    .reg x,y,p,s
    loop : .trip 20
    LDH *px++, x
    LDH *py++, y
    MPY x,y,p
    ADD s,p,s
    SUB n,1,n
    [n] B loop
    .return s
    .endproc

```

TAB. 1 – produit scalaire en assembleur linéaire

- un fichier `dotProduct_SA.h` doit aussi être défini et doit contenir les informations nécessaires lors de l'édition des liens : `.global dotProduct_SA`.
- Les directives `.cproc` et `.endproc` marquent le début et la fin d'une routine appellable depuis le C (ici `dotproduct` sans le préfixe "_").
- les paramètres qui suivent le nom de la routine correspondent aux paramètres de la fonction C. Notez qu'il n'y a pas de type (`px` correspondant à `int16 *pX`, `py` à `int16 *pY` et `n` la taille des vecteurs)
- `.reg` indique les variables locales utilisées par la routine. Notez qu'il n'est pas nécessaire d'associer un numéro de registre à chaque variable
- la boucle est définie par un label `loop` : et le branchement se fait tant que `n` n'est pas nul.
- il est possible de guider le compilateur en indiquant une valeur minimale d'itération de la boucle : c'est la directive `.trip`
- le chargement des données se fait via l'instruction `LDH` (*LoaD Half*) qui charge une valeur 16 bits dans une variable.
- s'il est nécessaire de retourner une valeur, cela se fait via l'instruction `.return`.

L'assembleur linéaire est donc un assembleur relativement haut niveau et proche du C :

- il n'est pas nécessaire d'associer un registre à chaque variable, ce qui permet au compilateur/assembleur d'optimiser l'allocation des registres.
- il n'est pas nécessaire d'écrire les instructions en parallèle, l'assembleur s'en charge.
- la gestion des tableaux est très simple puisque l'écriture C est permise (`*ptr++`).
- il n'est pas non plus nécessaire de désigner explicitement sur quelle unité de calcul les opérations arithmétiques se feront, ni le côté de DSP (1 ou 2).

Pour chaque version C écrire son équivalent en assembleur linéaire, analysez les tailles de cours de boucle et concluez. Afin qu'il n'y est pas de confusion entre les fonctions C et les routines asm, écrire une routine par fichier, le nom de la routine et le nom du fichier seront identiques, ce sera le nom de la fonction C avec le suffixe `_SA`, par exemple, si la fonction C est `dotProduct_B`, la routine asm sera `_dotProduct_B_SA` et le nom du fichier sera `dotProduct_B_SA.sa`.

2 filtrage

Soit le filtre FIR dont le code est donné table 2

```

#define N 1024
#define M 4
int16 X[N+M-1];
int16 H[M];
int16 Y[N];
void fir (int16 *pX, int16 *pH, int16 *pY, int n, int m, int s)
{
    int i, k;
    int32 y0;
    int16 round = 1 << (s-1); // arrondi pour calcul en virgule fixe
    for(i=0; i<n; i++) {
        y0 = round;
        for(k=0; k<m; k++) {
            y0 += pX[i+k] * pH[k];
        }
        pY[i] = (int16) (y0 >> s);
    }
}

```

TAB. 2 – Filtre FIR

2.1 Partie théorique

Le filter étant de taille fixe (4 coefficients), on décide de faire disparaître la boucle interne. Ecrire les versions 16-bits pour ce filtre à 4 coefficients avec les optimisations suivantes :

- version G (basique `fir_G(int16 *pX, int16 *pH, int16 *pY, int n, int m, int s)`): C classique avec la boucle interne sur les coefficients du filtre
- version B (basique `fir4_B(int16 *pX, int16 *pH, int16 *pY, int n)`): C classique
- version ILU2 (`fir4_ILU2(int16 *pX, int16 *pH, int16 *pY, int n)`): Loop Unrolling Interne d'ordre 2. A quoi faut il faire attention? Faire un schéma.
- version MPY (`fir4_MPY(int16 *pX, int16 *pH, int16 *pY, int n)`): utilisation des *intrinsic* `_mpy` (low x low) `_mpyh` (high x high) `_mpylh` (low x high) `_mphi` (high x low). Pour cela, les pointeurs sur tableaux sont castés en `int` afin de gérer des chargements 32 bits (donc de 2 `int16` 16-bits). Combien y a t il d'itération à faire dans la boucle?
- version DOPT (`fir4_DOTP(int16 *pX, int16 *pH, int16 *pY, int n)`): utilisation de l'*intrinsic* `_dotp2`.

2.2 Partie pratique

Implanter en C puis en assembleur linéaire ces différentes versions. Comparez les résultats aux résultats théoriques, puis les comparer aux fonctions de la librairie DSP.

3 Chronométrie

Il existe différentes façons de mesurer le temps écoulé sur DSP. La version retenue est de faire appel, via DSP/BIOS du timer temps réel haute résolution : `CLK_gettime()` définie dans `<clk.h>`. On prendra soin d'éliminer l'overhead de temps lié à l'appel de la fonction de chronométrie. En fonction des optimisations utilisées par le compilateur, il peut arriver qu'il soit impossible de chronométrer une fonction. Cela est dû au fait que le compilateur se rend compte que les calculs faits dans une fonction ne sont pas utilisés par la suite. Une solution est :

- étape #1 : récupérer le résultat dans une variable puis l'afficher,
- étape #2 : accumuler les résultats dans une variable puis l'afficher

La seconde étape est nécessaire dans certains cas, car le compilateur se rend compte que c'est *toujours le même calcul* qui est fait. Il décide donc de le faire qu'une seule fois.

Dans l'exemple suivant (code 3), les paramètres de la fonction `f` sont modifiés afin que le compilateur ne puisse pas appliquer certaines optimisations.

```
t0 = CLK_gethetime();
for(i=1; i<n; i++)
    r += f(a,b+i);
t1 = CLK_gethetime();
dt = t1-t0;
LOG_printf("result = %d", r);
LOG_printf("time = %d", dt);
```

TAB. 3 – chronométrie