Programmation Parallèle sur CPU Multicoeurs: Optimization using NEON

Travaux Pratiques

Préambule

Ces TPs, de trois heures chacun, s'adressent aux étudiants de l'option SMTR (Systèmes Multimédias et Technologies Réseaux) de la cinquième année du département ETN (Électronique et Technologies Numériques) de l'école polytechnique de l'université de Nantes. Ils ont pour objectif de leur fournir les bases nécessaires à la prise en main et à l'implantation d'algorithmes de traitement du signal sur une carte de type "raspberry" avec un processeur ARMv8-A . Après une brève description du kit de développement utilisé, le premier TP permet la prise en main de cet outil à travers deux exemples : le calcul de la somme de deux vecteurs et le calcul du produit scalaire. Le deuxième TP s'intéresse aux méthodes d'optimisation et à leur comparaison à travers l'implantation d'une application de filtrage numérique non récursif. Le troisième et dernier TP traite d'une application incluant un filtrage numérique récursif et un calcul de la densité spectrale de puissance du signal filtré.

L'environnement de développement

Introduction

Chacun des trois TPs proposés dans ce polycopié, vise la conception d'un programme et son implantation sur un processeur. L'environnement de développement utilisé par l'ensemble des trois TPs regroupe la carte Raspberry PI3 intégrant le processeur ARMv8-A cortex A-53 et fonctionnant sous ubuntu Mate, un compilateur C pour exploiter le coprocesseur NEON, gprof pour le profilage et gnuplot pour la visualisation des données.

La carte Raspberry Pi3

FIGURE 1 – Carte RPI3

Les principales caractéristiques techniques du RPI3 :
- SoC : Broadcom BCM2837 64 bits quadruple coeur
- CPU : 4 x ARM Cortex-A53, 1.2GHz
- RAM 1 Go
- BCM43438 avec WiFi et Bluetooth Low Energy (BLE) intégrés
- Port GPIO étendu de 40 broches
- 4 x USB 2 ports
- Sortie stéréo 4 pôles et port vidéo composite
- HDMI pleine taille
- Port de caméra CSI pour connecter la caméra Raspberry Pi
- Port d'affichage DSI pour connecter l'écran tactile du Raspberry Pi
- Port Micro SD pour le chargement de votre système d'exploitation et le stockage des données.

Observation des données : gnuplot

Gnuplot est un outil de visualisation de données. Il permet de tracer des graphiques 2D ou 3D. Il est disponible sur de nombreuses plates-formes. Les instructions peuvent sont à transmettre directement à l'invite.
Exemple des principales commandes :

- Lancement de gnuplot : taper "gnuplot"
- Sortie de gnuplot : taper "exit"
- Tracé d'une fonction : taper "plot" et "help plot"' pour avoir de l'aide sur la syntaxe. 3 modes de tracé :
  - Mode cartésien : taper "plot" suivi de la définition de la fonction. Exemple : plot 2*x
  - Mode paramétrique : taper "set parametric" puis la définition de la fonction. Exemple : plot 2*sin(t)*log(t)
  - Mode polaire : taper "set polar" puis la définition de la fonction. Exemple : plot 2*sin(t)*cos(t)
- Fonctions prédéfinies : en général celles acceptées par le langage C (ou autre)

Fonction	Résultat
abs(x)	valeur absolue de x, |x|
acos(x)	arc-cosinus de x
asin(x)	arc-sinus de x
atan(x)	arc-tangente de x
cos(x)	cosinus de x, x est en radians.
cosh(x)	cosinus hyperbolique de x, x est en radians
erf(x)	fonction erreur de x
exp(x)	fonction exponentielle de x, base e
inverf(x)	inverse de la fonction erreur de x
invnorm(x)	inverse de la distribution normale de x
log(x)	log de x, base e (i.e. ln(x))
log10(x)	log de x, base 10
norm(x)	distribution gaussienne normale
rand(x)	générateur de nombres pseudo-aléatoires
sgn(x)	1 si x > 0, -1 si x < 0, 0 si x=0
sin(x)	sinus de x, x est en radians
sinh(x)	sinus hyperbolique de x, x is en radians
sqrt(x)	racine carrée de x
tan(x)	tangente de x, x est en radians
tanh(x)	tangent hyperbolique de x, x est en radians

- Tracé de plusieurs fonctions dans la même fenêtre : taper "replot" suivi de la deuxième fonction à tracer.
- Tracé à partir d'un fichier de données (fichier texte) : "plot 'fichier.txt' with lines"

Profilage : gprof

Lorsque les programmes croissent en taille et en complexité, il est de plus en plus difficile d'isoler les problèmes qui sont la source des mauvaises performances. Le profiler analyse les

temps d'exécution des programmes pour éliminer les gouleaux d'étranglement. Par exemple, il peut déterminer le nombre de cycles passés dans l'exécution d'une fonction et le nombre de fois où elle est appelée. Ayant déterminé les sections critiques d'un programme, il est alors possible de mieux cibler les efforts d'optimisation. Pour les besoins des TPs, l'outil de profilage GNU 'gprof' sera utilisé. 3 étapes sont nécessaires pour sa mise en oeuvre :

1. Activer le profilage au moment de la compilation avec l'option '-pg'
   $ gcc -pg essai_gprof.c -o essai_gprof
2. Exécuter le fichier binaire ainsi produit
   $ ./essai_gprof
   Le fichier 'gmon.out' est alors généré.
3. lancer l'outil gprof
   $ gprof essai_gprof gmon.out > fichier_analyse.txt
   Le fichier 'fichier_analyse.txt' est alors généré. Il ne reste plus qu'à l'éditer pour avoir les informations de profilage.

Un exemple d'un tel fichier est donné par la figure 2.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 98.18      0.54     0.54  1000000   540.00   540.00  add_cpu
  1.82      0.55     0.01                             main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
        else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
        function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
        the function in the gprof listing. If the index is
        in parenthesis it shows where it would appear in
        the gprof listing if it were to be printed.


Copyright (C) 2012-2014 Free Software Foundation, Inc.
```

FIGURE 2 – Exemple d'information de profilage

Optimisations : Compilateur et options utilisées

Sans option d'optimisation, le compilateur cherchera à réduire le coût de la compilation pour permettre au débogage de produire les résultats escomptés. En activant l'optimisation (-O), le compilateur cherchera à améliorer la performance et/ou la taille du code au détriment du temps de compilation.
Il y a 4 niveaux d'optimisation :

- '-O0' : pas d'optimisation (par defaut).
- '-O1' : Optimisation modérée pour ne pas dégrader le temps de compilation.
- '-O2' : Optimisation complète, génère un code hautement optimisé et le plus lent temps de compilation.
- '-O3' : même optimisation que -O2 plus essai de vectorisation des boucles.

D'autres options permettent au compilateur d'utiliser des instructions NEON. Celles susceptibles d'être utilisées ici sont :

- '-std=c99' : La norme C99 introduit quelques nouvelles fonctionnalités qui peuvent être utilisées pour l'optimisation de NEON.
- '-mcpu=cortex-a9' : spécifie le nom du processeur cible ARM afin de déterminer les instructions appropriées.
- '-mfpu=neon' : pour utiliser le coprocesseur NEON.
- '-ftree-vectorize' : effectue la vectorisation des boucles (activé au niveau '-O3').

TP1 : Exemples d'implantation et évaluation des performances

Objectifs

Les objectifs, simples, de ce premier TP sont :

1. Prise en main du RPI3.
2. Prise en main de l'environnement.
3. Implantation d'algorithmes simples.
4. Évaluation des performances avant et après optimisation.

Deux applications serviront de base pour atteindre ces objectifs : l'implantation d'une simple addition de vecteur et l'implantation d'un produit scalaire.

1^{ere} Application : Somme de deux vecteurs

La première application visée par ce premier TP est celle qui calcule la somme de deux vecteurs.

```
#include <stdio.h>
#include <arm_neon.h>
#define COUNT 8
void add_cpu(int8_t v1[COUNT], int8_t v2[COUNT], int8_t vres[COUNT])
{
int i;
for (i=0; i<COUNT; i++) vres[i] = v1[i]+v2[i];
}
main()
{
int8_t v1_vec[COUNT] = {1,2,3,4,5,6,7,8};
int8_t v2_vec[COUNT] = {8,7,6,5,4,3,2,1};
int8_t vres_vec[COUNT];

add_cpu(v1_vec, v2_vec, vres_vec);
}
```

Travail à faire

Utilisation du processeur coeur

1. Écrire et tester ce programme d'addition de 2 vecteurs.
2. Profiler ce programme en mesurant le temps mis par le processeur pour exécuter cette fonction (add_cpu) N fois (boucle for avec le nombre d'itérations N = 10^6 par exemple). Utiliser gprof (option -pg au moment de la compilation).

Optimisation par vectorisation

Vectorisation lors de la compilation

1. Commencer par inclure le fichier header "arm_neon.h" dans votre code C.
2. Écrire une nouvelle fonction "add_vectorization_c" que le compilateur peut vectoriser d'une manière optimale. Pour cela il faut :
   - utiliser le mot clé "__restrict" pour garantir au compilateur que les pointeurs v1 et v2 n'adressent pas des blocs mémoire qui se chevauchent,
   - forcer la boucle à toujours exécuter un multiple de 4 itérations en masquant les deux derniers bits de n pour le test de fin de course :
     for (i=0 ; i < (COUNT & $\sim$ 3) ; i++) v3[i] = v1[i] + v2[i] ;
3. Compiler en rajoutant à vos options de compilation les options "-mfpu=neon" et "-ftree-vectorize".
4. Moyennant gprof, reprofiler les deux fonctions add_cpu et add_vectorization_c et commenter les résultats obtenus.

Vectorisation manuelle

1. Ajouter une fonction "add_vectorization_m" dans votre code C qui vectorise manuellement le calcul de la somme. Pour cet exemple simple on se contentera d'entiers sur 8 bits (int8_t). Au lieu d'utiliser un seul accumulateur comme dans le cas de "add_cpu", la vectorisation manuelle consistera à utiliser 8 accumulateurs en parallèle.
2. Dans le main, les vecteurs v1, v2 et v3 sont à définir comme des vecteurs de type int8_t.
3. Compiler avec les mêmes options que précédemment. La dernière option "-ftree-vectorize" n'est pas utile à ce niveau puisque la vectorisation est manuelle.
4. Profiler les 3 fonctions et commenter.
5. Changer la taille des vecteurs (COUNT = 256 par exemple) et refaire le même travail de profilage et d'optimisation.

Optimisation par l'utilisation des NEON intrinsics

La procédure à suivre dans ce simple exemple d'addition de vecteurs, consiste à d'abord charger les données de la mémoire dans les registres et ensuite additionner le contenu de ces registres. Pour cela il faut :

1. Écrire une fonction "add_neon" avec des pointeurs sur les vecteurs d'entrée v1 et v2 de type int8_t.
2. Dans cette fonction, commencer par déclarer 3 registres de type (taille) int8x8_t (vec_neon1, vec_neon2 et vec_neon3 par exemple).
3. Charger les données des vecteurs d'entrée v1 et v2 dans ces registres moyennant l'instruction "vld1_s8" (http ://infocenter.arm.com/help).
4. Ajouter le contenu de ces deux registres moyennant l'instruction "vadd_s8" et stocker le résultat dans vec_neon3.
5. Compiler avec les mêmes options que précédemment et tester cette nouvelle fonction seule (vérifier qu'elle fournit le bon résultat).
6. Profiler les 4 fonctions dans le cas d'une boucle for avec le nombre d'itérations N = 10^6 et commenter les résultats obtenus.

2^{eme} Application : Produit Scalaire

La deuxième application visée par ce premier TP est celle qui calcule le produit scalaire de deux vecteurs.

```
#include <stdio.h>
#include <stdlib.h>
#define N 1024
void data(int a[N], int b[N])
{
int i;
for (i=0; i<N; i++) { a[i] = i; b[i]=i; }
}
int dotproduct_cpu(int *a, int *b, int n)
{
int i;
int sum = 0;
for (i=0; i<n; i++) sum += a[i]*b[i];
return sum;
}
main()
{
int result_cpu;
int v1[N], v2[N];
data(v1, v2);
result_cpu = dotproduct_cpu(v1, v2, N);
}
```

Comme pour l'addition de vecteurs, nous allons ici profiler d'abord la fonction "dotproduct_cpu" et ensuite chercher à l'optimiser.

Travail à faire

Utilisation du processeur coeur

1. Écrire et tester ce programme.
2. Profiler ce programme en mesurant le temps mis par le processeur pour exécuter cette fonction (dotproduct_cpu) N fois (N = 50000 par exemple).

Optimisation

Déroulement manuel des boucles

1. Inclure le fichier header "arm_neon.h" dans votre code C.
2. Écrire une nouvelle fonction "dotproduct_vecman" qui va chercher à dérouler manuellement la boucle. Pour cela, il faut
   - commencer par utiliser le mot clé "__restrict" pour garantir au compilateur que les pointeurs a et b n'adressent pas des blocs mémoire qui se chevauchent,
   - redéfinir le type de la variable sum et l'initialiser comme suit
     int32_t sum[4] = {0, 0, 0, 0} ;
     qui signifie que sum est un vecteur de 4 éléments de 32 bits chacun (4 accumulateurs pouvant être traités simultanément par 4 "lanes").
   - forcer la boucle à toujours exécuter un multiple de 4 itérations en masquant les deux derniers bits de n pour le test de fin de course :
     for (i=0 ; i < (n & $\sim$ 3) ; i +=4)
     i +=4 signifie qu'à la première itération, les 4 premiers produits sont calculés et stockés dans chacun des 4 éléments de la variable sum. L'itération suivante calculera et accumulera les 4 produits suivants et ainsi de suite. A la sortie de la boucle, les 4 accumulateurs doivent être additionnés.
3. Compiler en rajoutant à vos options de compilation les options "-mfpu=neon" et "-ftree-vectorize".
4. Moyennant gprof, reprofiler les deux fonctions dotproduct_cpu et dotproduct_vecman dans le cas d'une boucle for avec le nombre d'itérations N = 50000 et commenter les résultats obtenus.

Utilisation des NEON intrinsics

Comme précédemment, la procédure consiste à d'abord charger les données de la mémoire dans les registres, a faire ensuite une multiplication et une accumulation et à extraire enfin les données des registres pour les stocker dans la variable à renvoyer. La démarche peut être :

1. Écrire une fonction "dotproduct_intrinsics" avec des pointeurs (utilisant "__restrict") sur les vecteurs d'entrée a et b de type int16_t et qui renvoie un vecteur de type int32_t.
2. Dans cette fonction, commencer par déclarer 2 vecteurs a_i et b_i de type int16x4_t, un vecteur de type int32x4_t qu'il faut initialiser (sum_i = {0, 0, 0, 0} par exemple), un vecteur pour le calcul intermédiaire de type int32x2_t de dimension 2 (tmp[2]), la

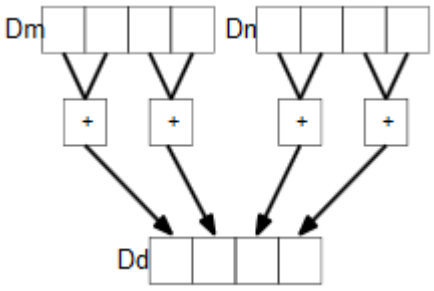

FIGURE 3 – Exemple d'opération vpadd (données de type int16_t)

variable de sortie de type int32_t (sum) et enfin la variable pour l'indice d'incrémentation de type int16_t (i par exemple).

3. Ouvrir une boucle for comme précédemment pour parcourir les éléments des tableaux for (i=0 ; i < (n & $\sim$ 3) ; i +=4)
4. Charger les données pointées par a et b dans les registres a_i et b_i moyennant l'instruction "vld1_s16" (NB : ce sont les 4 premières valeurs de chacun des vecteurs qui sont chargées dans chacun des registres a_i et b_i.
5. Multiplier les contenus de ces deux registres et les accumuler moyennant l'instruction "vmlal_s16" (le résultat de l'addition et mis dans sum_i ainsi que le cumul).
6. A la sortie de la boucle, les 2 premières (hautes) valeurs (sur 32 bits chacune) de sum_i sont mises dans tmp_i[0] (2 valeurs de 32 bits) et les deux dernières (basses) valeurs sont mises dans tmp_i[1] moyennant les instructions "vget_high_s32" et "vget_low_s32".
7. l'étape suivante consiste d'abord à ajouter les paires adjacentes des éléments des deux vecteurs tmp_i[0] et tmp_i[1] et à placer les résultats dans le vecteur de destination tmp_i[0] moyennant les instructions "vpadd_s32" (voir figure 3). Il faut ensuite faire la même chose avec les deux vecteurs de tmp_i[0] et stocker la somme dans tmp_i[0].
8. Pour finir, il suffit de récupérer la valeur stockée dans la première voie (lane) de tmp_i[0] (moyennant "vget_lane_s32") et de la stocker dans la variable sum (variable retournée par la fonction).
9. Dans le main, et avant l'appel de la fonction, vous avez besoin de redéclarer la variable de sortie en type int32_t.
10. Compiler avec les mêmes options que précédemment et tester cette nouvelle fonction seule (vérifier qu'elle fournit le bon résultat).
11. Profiler les fonctions dans le cas d'une boucle for avec le nombre d'itérations N = 50000 et commenter les résultats obtenus.

TP2 : Implantation et optimisation de filtres FIR

Objectifs

Les filtres numériques sont utilisés dans beaucoup d'applications de traitement du signal. L'objectif de ce second TP est d'optimiser l'implantation d'un filtre FIR en exploitant les capacités de traitement parallèle (SIMD) de l'architecture des processeurs ARM-A.

Application

L'application très simple à développer consiste à générer un bruit blanc et à le filtrer par un filtre FIR passe-bas demi-bande.

Générateur de bruit

Le générateur de bruit blanc génère un bruit équiréparti entre -500 et 500 de type int16_t. On pourra utiliser la fonction C rand() qui génère un nombre compris entre 0 et $RAND\_MAX$-1 (inclure <stdlib.h>).

Filtre

Le filtre utilisé est un filtre non récursif de longueur 8.

$$H(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4} + b_5 z^{-5} + b_6 z^{-6} + b_7 z^{-7}.$$

Les coefficients de ce filtre sont :

$$b_0 = 1, \quad b_1 = -2, \quad b_2 = 4, \quad b_3 = -2, \quad b_4 = -2, \quad b_5 = 4, \quad b_6 = -2, \quad b_7 = 1$$

Le code C incomplet de cette application peut s'écrire :

```
#include <stdio.h>
#include <stdlib.h>
#define Nc 8
#define Ne 8192
int16_t Gen_data()
```

```
{
return (rand().....) ; //à compléter
}
void fir_cpu(int16_t *in, const int16_t *coef, int32_t *out, const uint16_t ncoef, const
uint16_t nech)
{
uint16_t i,j ;
int32_t result = 0 ;
for (i=0 ; i<nech ; i++) out[i] = 0 ; //initialisation
for (i=0 ; i<ncoef ; i++) out[i] = in[i] ; // premiers échantillons de sortie sont pris égaux
à ceux d'entrée
for (i=ncoef ; i<nech ; i++)
{
result = 0 ;
for (j=0 ; j<ncoef ; j++)
{
result += ..... ; // à compléter
}
out[i] = .... ; // à compléter
}
}
int main()
{
uint16_t i ;
const uint16_t cof[] = {1, -2, 4, -2, -2, 4, -2, 1} ;
int16_t xin[Ne] ;
int32_t yout[Ne] ;
// Créer le vecteur d'entrée xin en appelant la fonction Gen_data
..... à compléter
// Filtrer les données d'entrée en réitérant le processus 10000 fois
..... à compléter
return 0 ;
}
```

Comme pour le TP1, nous allons ici profiler d'abord la fonction "fir_cpu" et ensuite chercher à l'optimiser.

Travail à faire

Utilisation du processeur coeur

1. Écrire, compléter et tester ce programme (en comparant le résultat fourni avec celui de Matlab par exemple).
2. Profiler ce programme en mesurant le temps mis par le processeur pour exécuter cette fonction (fir_cpu) N fois (N = 10000 par exemple).

Optimisation

Outre les boucles d'initialisation, la fonction fir_cpu exploite deux boucles : une boucle externe d'indice i qui parcourt le nombre d'échantillons et une boucle interne d'indice j qui parcourt le nombre de coefficients. Dans ce TP, on va chercher à dérouler l'une puis l'autre puis l'une et l'autre.

Déroulement manuel de la boucle interne

1. Inclure le fichier header "arm_neon.h" dans votre code C.
2. Écrire une nouvelle fonction "fir_vec_in" qui va chercher à dérouler manuellement la boucle interne. Pour cela, il faut
   - redéfinir le type de la variable result en "int32_t sum[4];" qui signifie que result est un vecteur de 4 éléments de 32 bits chacun (4 accumulateurs pouvant être traités simultanément par 4 "lanes").
   - initialiser cette variable après le traitement de chaque échantillon
     result[0] = result[1] = result[2] = result[3] = 0;
   - forcer ensuite la boucle interne à toujours exécuter un multiple de 4 itérations :
     for (j=0; j < ncoef; j +=4)
     j +=4 signifie qu'à la première itération, les 4 premiers produits sont calculés et stockés dans chacun des 4 éléments de la variable result. L'itération suivante calculera et accumulera les 4 produits suivants et ainsi de suite. A la sortie de la boucle interne, les 4 accumulateurs doivent être additionnés et stockés dans la variable out[i].
3. Compiler en rajoutant à vos options de compilation les options "-mfpu=neon" et "-ftree-vectorize".
4. Vérifier le bon fonctionnement.
5. Moyennant gprof, reprofiler les deux fonctions fir_cpu et fir_vec_in dans le cas N = 10000 et commenter les résultats obtenus.

Déroulement manuel de la boucle externe

1. Écrire une nouvelle fonction "fir_vec_out" qui va chercher à dérouler manuellement la boucle externe. Pour cela, il faut
2. forcer la boucle externe à toujours exécuter un multiple de 4 itérations :
   for (i=ncoef; i < ncoef; i +=4)
   j +=4 signifie qu'à la première itération, les 4 premiers échantillons de sortie (out[i], out[i+1], out[i+2] et out[i+3]) sont calculés simultanément. L'itération suivante calculera et accumulera les 4 échantillons suivants et ainsi de suite.
3. Compiler l'ensemble et vérifier le bon fonctionnement de cette nouvelle fonction.
4. Moyennant gprof, reprofiler les 3 fonctions fir_cpu, fir_vec_in et fir_vec_out dans le cas N = 10000 et commenter les résultats obtenus.

En général dans le cas de boucles imbriquées, le déroulement d'une seule boucle apporte peu en termes de performance. Dans la suite on se propose de dérouler et la boucle interne et la boucle externe.

Déroulement manuel des 2 boucles

1. En vous inspirant de ce qui a été fait pour chacune des boucles, écrire une nouvelle fonction fir_vec_inout qui permet de dérouler les deux boucles.
2. Compiler et tester cette nouvelle fonction.
3. Reprofiler les 4 fonctions fir_cpu et fir_vec_in dans le cas N = 10000 et commenter les résultats obtenus.
4. Recompiler l'ensemble en ajoutant aux options habituelles l'option -O2 qui permet d'optimiser le code C. Commenter les résultats.

Utilisation des NEON intrinsics

Comme précédemment, la procédure consiste à d'abord charger les données de la mémoire dans les registres, a faire ensuite une multiplication et une accumulation et à extraire enfin les données des registres pour les stocker dans la variable à renvoyer. La démarche pour dérouler la boucle interne peut être :

1. Écrire une fonction "fir_intrinsics" ayant exactement les mêmes arguments que les fonctions précédentes.
2. Dans cette fonction, commencer par déclarer les indices de boucles i et j en type uint16_t. Déclarer ensuite un vecteur result de type int32x4_t, deux vecteurs pour le calcul intermédiaire de type int16x4_t (coefi et ini par exemple).
3. Comme précédemment, initialiser le vecteur de sortie à 0 (out[i]=0) et affecter ensuite les premières valeurs d'entrée à ce tableau.
4. Ouvrir une boucle for comme précédemment pour parcourir les éléments restants du tableau : for (i=ncoef ; i < nech ; i ++)
5. Initialiser à 0 la variable result moyennant l'instruction "vdupq_n_s32(0)".
6. Ouvrir la boucle pour le parcours des coefficients : for (j=0 ; j < ncoef ; j +=4)
7. Charger les données pointées par le vecteur d'entrée in dans le registre intermédiaire ini moyennant l'instruction "vld1_s16" (NB : ce sont les 4 premières valeurs qui sont chargées).
8. Faire exactement la même chose pour les coefficients (4 coefficients sont chargés dans coefi).
9. Multiplier les contenus de ces deux registres et les accumuler moyennant l'instruction "vmlal_s16" (le résultat de l'addition et mis dans result ainsi que le cumul).
10. A la sortie de la boucle, l'échantillon de sortie out[i] s'obtient en ajoutant le contenu des 4 voies (lanes) du registre result moyennant l'instruction "vgetq_lane_s32(result, lane number).
11. Dans le main, et avant l'appel de la fonction, vous avez besoin de redéclarer la variable de sortie en type int32_t.
12. Compiler avec les mêmes options que précédemment et tester cette nouvelle fonction seule (vérifier qu'elle fournit le bon résultat).
13. Profiler les fonctions dans le cas d'une boucle for avec le nombre d'itérations N = 50000 et commenter les résultats obtenus.

TP3 : Filtrage numérique récursif et mise en oeuvre d'une application d'estimation de la densité spectrale de puissance

Objectifs

Le contexte de ce dernier TP est une application simple d'estimation de la Densité Spectrale de Puissance (DSP) d'un signal filtré. L'objectif est d'optimiser l'implantation du filtre récursif et de programmer le calcul de la DSP.

Application

L'application à développer est donnée par la figure 4

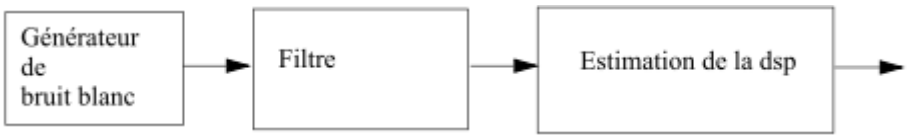

FIGURE 4 – Estimation de la DSP

Générateur de Bruit

Le signal de bruit utilisé ici est le même que celui utilisé dans le TP2 ; à savoir un signal équiréparti entre -500 et 500 et de type int16_t.

Filtre récursif

Il s'agit ici de faire l'implantation d'un filtre récursif du second ordre. La fonction de transfert en Z d'un tel filtre est donnée par

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

. Le filtre choisi est un réjecteur à 50 Hz. Pour une fréquence d'échantillonnage de 1000Hz, les coefficients de ce filtre sont

- $a_1 = 1.8452$,

- a_2 = -0.9392,
- b_0 = 1.0,
- b_1 = -1.9031,
- b_2 = 1.0.

Estimation de la dsp

L'estimation de la DSP s'effectuera avec la méthode du périodogramme. Soit

$$X_k = \left[x_{kN}, ..., x_{(k+1)N-1}\right] \qquad k = 0, ..., M-1$$

où x_n est l'échantillon du signal filtré à l'instant n et N la longueur des blocs (N=1024). On calcule la FFT :

$$Y_k = FFT\left(X_k\right) = \left[y_{kN}, ..., y_{(k+1)N-1}\right] \qquad k = 0, ..., M-1.$$

Enfin on estime la DSP sur M blocs (où M=100) :

$$\phi_n = \frac{1}{M}\sum_{k=0}^{M-1} \left|y_{kN+n}\right|^2 \qquad n = 0, ..., N-1.$$

Ce qui revient à moyenner le module au carré des M blocs Yk.

Travail à faire

Utilisation du processeur coeur

1. Écrire le programme et visualiser graphiquement l'estimation de la dsp obtenue. Pour la FFT, on pourra utiliser les deux fichiers calcul.h et calcul.c (disponibles sur la page document du site du département).
2. Pour la compilation il vous faut (à travers un makefile peut être)
   - compiler d'abord le fichier calcul.c "gcc -c calcul.c -lm",
   - compiler ensuite le fichier iir que vous venez d'écrire avec les options habituelles,
   - compiler enfin les deux fichiers .o crées "gcc calcul.o iir.o iir -lm -pg"
   - lancer l'exécutable iir "./iir"
3. Profiler ce programme en mesurant le temps mis par le processeur pour exécuter cette fonction (iir_cpu) N fois (N = 10000 par exemple).

Optimisation

Comme pour les précédents TPs proposer une solution pour optimiser la fonction iir.