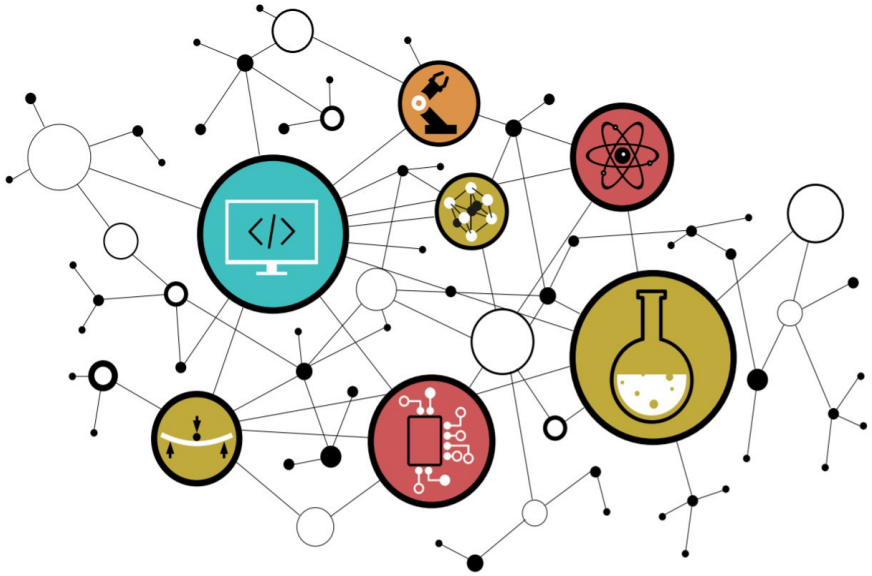
TRAVAUX PRATIQUES

MEMOIRE CACHE
ET MEMOIRE SRAM ADRESSABLE

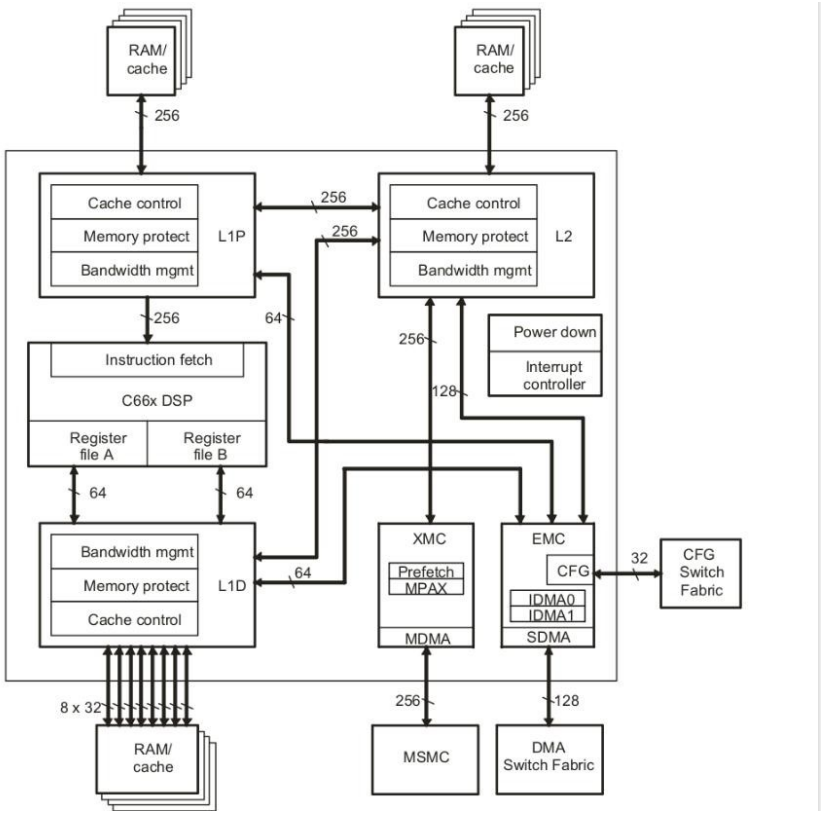


Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

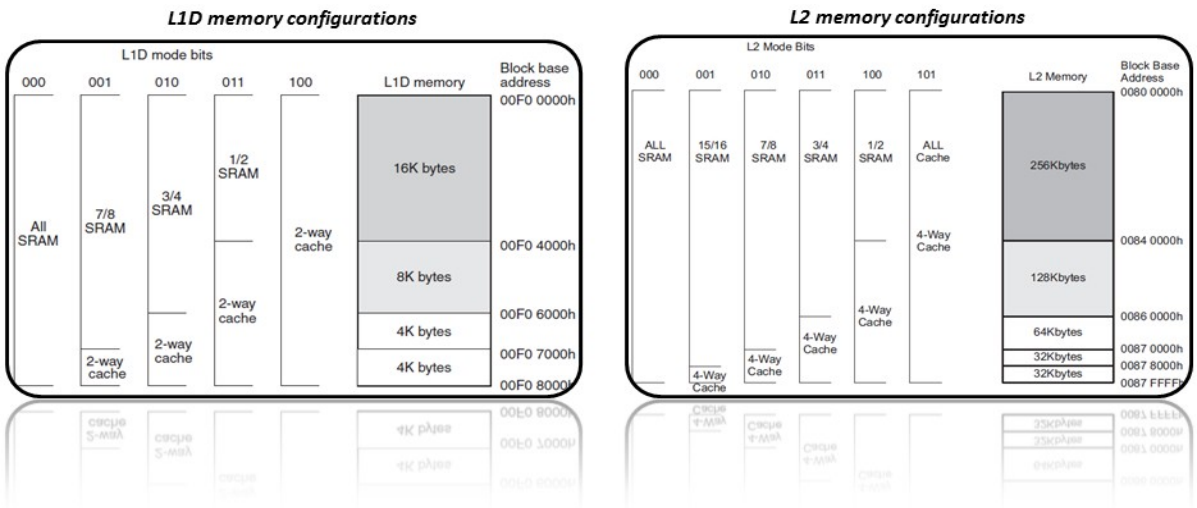
SOMMAIRE

4.1. Mémoire locale SRAM adressable

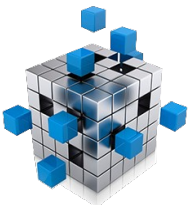
Nous allons maintenant chercher à passer outre les mémoires caches L1D et L2 en ne configurant qu'une partie des niveaux L1D et L2 en cache et le reste en mémoire SRAM adressable par octet. Ceci nous permettra de placer des données et le code souhaité (algorithme) dans des niveaux mémoire proches du CPU et ainsi de gagner en déterminisme (temps réel).

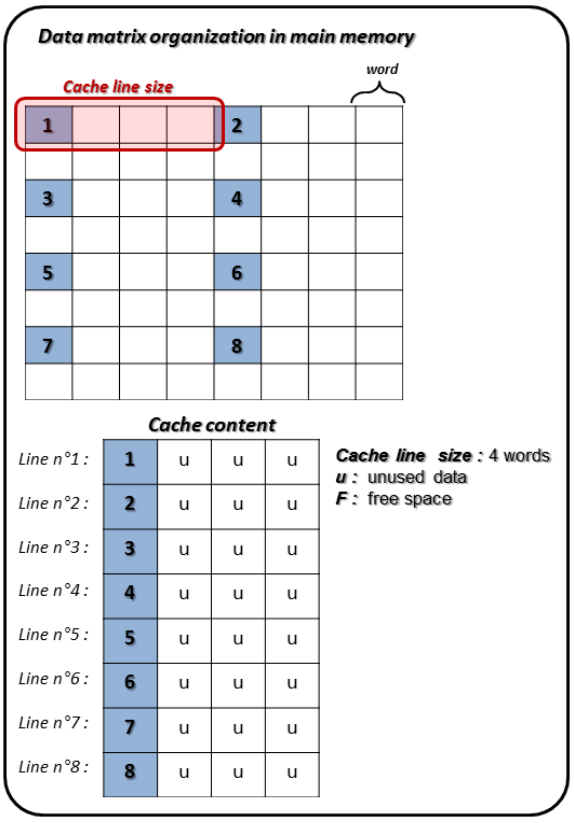


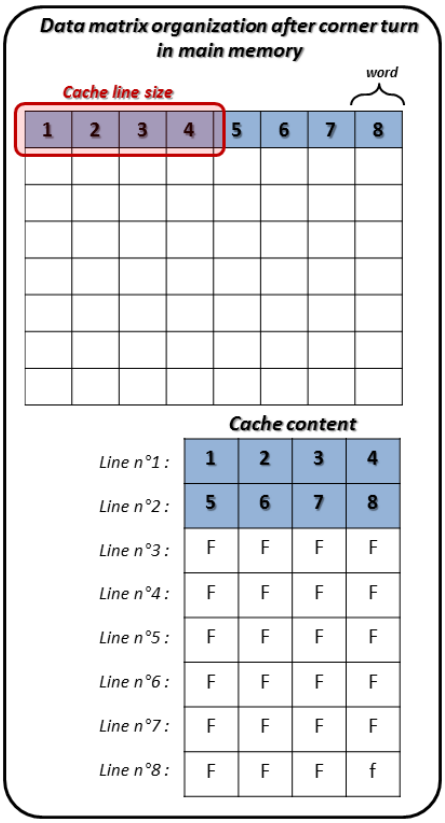
Dans l'exemple d'un produit scalaire, l'impact en performance restera minime. Néanmoins, pour grand nombre d'algorithmes du traitement numérique du signal (traitement d'image et d'antenne, etc), laisser opérer les contrôleurs cache sans ordonnancement des données avant traitement peut avoir une incidence énorme sur le temps d'exécution global de l'application. Dans l'exemple qui suit, nous pouvons observer une matrice de données manipulée par indexage (chargement/lecture donnée 1 puis 2 puis 3, etc). Rappelons également que les mémoires caches sont pilotées par des contrôleurs effectuant des copies d'informations depuis les niveaux hiérarchiques mémoire supérieurs. Ces contrôleurs implémentent des mécanismes de prédiction spéculatifs. Par exemple, chaque copie d'information se fait par ligne (ensemble d'octets).



Un cache étant chargé par ligne (principe de localité spatiale), nous sommes alors amenés à charger un grand nombre de données ''potentiellement'' inutilisées en cache. Une technique couramment utilisée, consiste à ré-implémenter l'algorithme de façon à traiter des données pré-ordonnancées en mémoire (technique du corner turn). Cette stratégie permet de diminuer le nombre d'accès au cache et assure une utilisation optimale des espaces de stockage en cache. Rappelons que l'accès aux ressources en mémoire est l'un des principaux goulots d'étranglement en termes de performance à notre époque. Nous avons également remarqué que notre architecture possède un grand nombre de registres de travail généralistes par CPU. L'objectif étant de charger les données locales à une fonction ou une boucle dans les registres du cœur, pour les traiter par la suite en minimisant au plus les allers-retours vers le cache. Écritures différées en mémoire en fin de traitement.

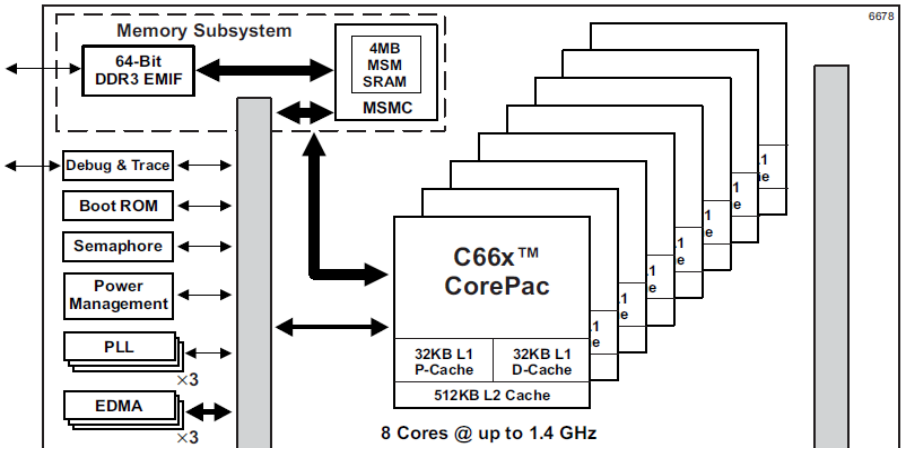






Dans notre cas, l'objectif est différent. L'algorithme d'un produit scalaire sans transformations mathématiques dans une optique de diminution de sa complexité, manipule les vecteurs d'entrée dans l'ordre de façon séquentielle. Cet exercice n'a donc pour objectif que d'illustrer la possibilité d'obtenir un espace de stockage adressable à accès rapide physiquement proche du cœur (L1D et L2). Nous maîtriserons alors avec certitude les données présentes aux niveaux L1D et L2 afin de garantir un déterminisme à l'exécution. Cette solution permet par exemple de s'affranchir localement du principe de corner turn précédemment présenté. En effet, nous allons manuellement effectuer le travail des contrôleurs de cache utilisant quant à eux des mécanismes de localité temporelle (LRU) pour leurs allocations/libérations de lignes de cache. Traitement impossible sur processeur généraliste GPP. Ceci permettra de rendre notre algorithmique quasiment insensible à la charge éventuelle du cache. Autre possibilité non explorée dans cette trame d'enseignement, la parallélisation des copies mémoire descendantes (DDR vers L1D), avec le calcul algorithmique (L1D et CPU) et avec les copies mémoire montantes (L1D vers DDR). Cette approche est implémentée par certains partenaires et offre des gains en performance considérables.

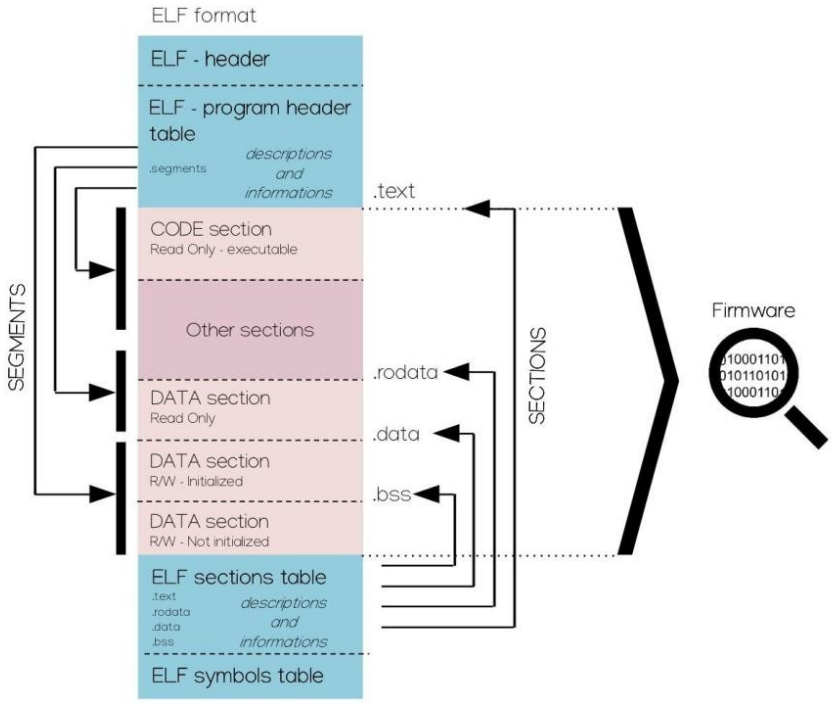
Nous avons possibilité de configurer les différents niveaux mémoire en cache ou en SRAM adressable de façon statique à la compilation ou dynamique à l'exécution. Rappelons avant tout ci-dessous l'architecture mémoire de notre processeur :



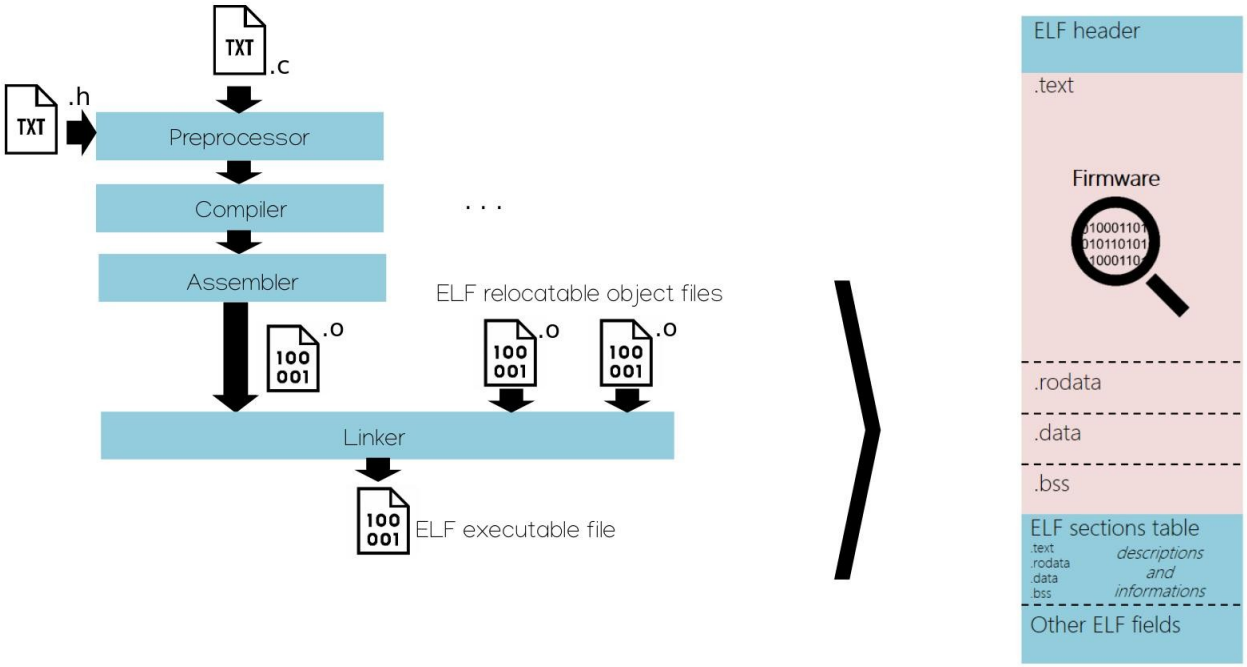
- Dans notre projet CCS, remplacer le script linker C6678_unified.cmd par le fichier /disco/c6678/test/map/C6678_unified_fir.cmd
- Analyser le fichier script Linker /disco/c6678/test/map/C6678_unified_fir.cmd

```
-stack 0x2000
-heap 0x2000

MEMORY
{
 LOCAL_L1P_SRAM:  origin = 0x00E00000 length = 0x00008000   /* 32kB LOCAL L1P/SRAM */
 LOCAL_L1D_SRAM:  origin = 0x00F00000 length = 0x00008000   /* 32kB LOCAL L1D/SRAM */
 LOCAL_L2_SRAM:   origin = 0x00800000 length = 0x00080000   /* 512kB LOCAL L2/SRAM */
 MSMCSRAM:        origin = 0x0C000000 length = 0x00400000   /* 4MB Multicore shared Memmory */

 EMIF16_CS2:     origin = 0x70000000 length = 0x04000000    /* 64MB EMIF16 CS2 Data Memory */
 EMIF16_CS3:     origin = 0x74000000 length = 0x04000000    /* 64MB EMIF16 CS3 Data Memory */
 EMIF16_CS4:     origin = 0x78000000 length = 0x04000000    /* 64MB EMIF16 CS4 Data Memory */
 EMIF16_CS5:     origin = 0x7C000000 length = 0x04000000    /* 64MB EMIF16 CS5 Data Memory */

 /* TMDSEVM6678L board specific */
 DDR3:           origin = 0x80000000 length = 0x20000000   /* 512MB external DDR3 SDRAM */
}

SECTIONS
{
 .text          >  MSMCSRAM
 .stack         >  MSMCSRAM
 .bss           >  MSMCSRAM
 .cio           >  MSMCSRAM
 .const         >  MSMCSRAM
 .data          >  MSMCSRAM
 .switch        >  MSMCSRAM
 .sysmem        >  MSMCSRAM
 .far           >  DDR3
 .args          >  MSMCSRAM
 .ppinfo        >  MSMCSRAM
 .ppdata        >  MSMCSRAM

 /* COFF sections */
 .pinit         >  MSMCSRAM
 .cinit         >  DDR3

 /* EABI sections */
 .binit         >  MSMCSRAM
 .init_array    >  MSMCSRAM
 .neardata      >  MSMCSRAM
 .fardata       >  DDR3
 .rodata        >  MSMCSRAM
 .c6xabi.exidx  >  MSMCSRAM
 .c6xabi.extab  >  MSMCSRAM

 /* project specific sections */

       /* user sections */
}
```

Rappelons (cf. cours 1A en Archi. des ordi.) que **les sections sont des zones de codes ou de données statiques allouées à la compilation et l'édition des liens, et présentes dans le fichier binaire exécutable de sortie (ELF, COFF, etc)**. Les sections seront par la suite mappées en mémoire physique du processeur. Dans notre cas, soit en DDR externe, soit en MSMC/L2/L1D/L1P interne.



- A quoi correspond la zone **MEMORY** du script ?
- En vous aidant de l'annexe ou de la documentation technique **disco/tp/doc/datasheet - optimizing compiler – spru187v.pdf**, préciser le rôle de la zone **SECTIONS** du script.
- Dans quelle section est rangée la **pile** ou stack système (allocations automatiques) ?
- Dans quelle section est rangé le **tas** ou heap système (allocations dynamiques) ?
- Dans quelle section est rangé le **code utilisateur** (s'aider de la page suivante) ?
- Analyser les parties du script permettant de configurer les niveaux mémoire L1D et L2 comme suit (à entourer sur la page précédente) :
    - **32Ko L1D** : 28Ko en L1D SRAM adressable et 4Ko en L1D cache
    - **512Ko L2** : 480Ko en L2 SRAM adressable et 32Ko en L2 cache
- Créer 3 nouvelles sections propres au projet et les placer tel que précisé ci-dessous :
    - **.ddrsdram** : section statique présente en DDR
    - **.l2sram** : section statique présente en L2 SRAM
    - **.l1dsram** : section statique présente en L1D SRAM
- Compiler puis exécuter le programme. Valider son bon fonctionnement.



Les allocations statiques représentent toutes les allocations de ressources mémoire réalisées à la compilation et l'édition des liens, et donc présentes dans le fichier binaire ELF de sortie. Les variables statiques admettent donc une existence sur un media de stockage de masse avant même l'exécution d'un programme en mémoire principale. Les références symboliques, ou adresses logiques, de chaque fonction et variable statique sont donc inchangées (statiques) durant la totalité de la vie d'un programme binaire sans nouvelle compilation et édition des liens du projet logiciel source. Contrairement aux variables locales (allocations automatiques ou dynamiques sur le segment de pile) et allocations dynamiques sur le segment de tas, pour lesquelles les allocations de ressources mémoire sont réalisées à l'exécution du programme dans des segments logiques dédiés (Pile ou Tas).

Une *section* est une zone logique du *firmware*. Un *Firmware* peut être également nommé micrologiciel en Français. Le *firmware* représente dans cet enseignement le code (forcément statique) et les données statiques binaires strictement utiles au fonctionnement d'un programme. Le *firmware* est quant à lui encapsulé dans un cartouche au format ELF (en-tête, table des sections, en-tête du programme, etc) proposant au noyau du système (Linux) une description et des informations sur le micrologiciel afin d'aider à sa manipulation (préparation des segments mémoire, association de propriétés et privilèges, etc). Sauf si un développeur créé explicitement de nouvelles sections en spécifiant des attributs spécifiques durant une déclaration d'une variable (ce que nous sommes en train de faire), une application pourra comporter au plus 4 sections applicatives par défaut afin de gérer l'ensemble des besoins standards en allocations statiques de ressources (les noms suivants sont hérités d'Unix) :

- .text (CODE - Read Only - executable) : section encapsulant le code binaire statique du programme
- .rodata (DATA - Read Only – not executable) : section encapsulant les données statiques accessibles en lecture seule
- .data (DATA - Read/Write – not executable) : section encapsulant les données statiques initialisées accessibles en lecture et écriture
- .bss (DATA - Read/Write – not executable) : section encapsulant les données statiques non-initialisées accessibles en lecture et écriture

4.2. Préchargement des données de DDR DRAM vers L2 SRAM

- A partir de maintenant et jusqu'à la fin de la trame de travaux pratiques, toutes nos futures optimisations (mémoire et périphériques d'accélération) utiliseront l'algorithme vectorisé fir_sp_opt_r4 précédemment développé et offrant un niveau optimum d'accélération.

- Remplacer le fichier main.c actuel par celui présent dans disco/c6678/sram_cache/main.c et analyser les déclarations des vecteurs statiques (variables globales) qui assureront des stockages partiels de nos vecteurs d'entrée et de sortie. Ces vecteurs stockeront temporairement en L2 SRAM et L1D SRAM des tronçons de vecteurs à traiter.

```
/* arrays allocations (bytes) :
* xk_sp (DDR)    |---------------- 256Kb or 4Mb --------------------|
* xk_sp_l2       |------- 128Kb + 256b - 4 ------| overloap
* xk_sp_l1d      |--- 8Kb + 256b - 4  ---| overloap
* a_sp_l1d         |- 256b -|
* yk_sp_l1d      |----- 8Kb -----|
* yk_sp_l2       |------------- 128Kb -------------|
* yk_sp (DDR)    |--------- (256Kb or 4Mb) - 256b + 4 ---------|
*/
float32_t xk_sp[XK_LENGTH];
float32_t yk_sp_ti[YK_LENGTH];
float32_t yk_sp[YK_LENGTH];
float32_t xk_sp_l2[L2_ARRAY_LENGTH + A_LENGTH - 1];
float32_t yk_sp_l2[L2_ARRAY_LENGTH];
float32_t xk_sp_l1d[L1D_ARRAY_LENGTH + A_LENGTH - 1];
float32_t yk_sp_l1d[L1D_ARRAY_LENGTH];
float32_t a_sp_l1d[A_LENGTH];

/* memory segmentation */
#pragma DATA_SECTION(xk_sp,".ddrsdram");
#pragma DATA_SECTION(yk_sp_ti,".ddrsdram");
#pragma DATA_SECTION(yk_sp,".ddrsdram");
#pragma DATA_SECTION(xk_sp_l2,".l2sram");
#pragma DATA_SECTION(yk_sp_l2,".l2sram");
#pragma DATA_SECTION(xk_sp_l1d,".l1dsram");
#pragma DATA_SECTION(yk_sp_l1d,".l1dsram");
#pragma DATA_SECTION(a_sp_l1d,".l1dsram");

/* arrays alignments - CPU data path length 64bits */
#pragma DATA_ALIGN(xk_sp, 8);
#pragma DATA_ALIGN(a_sp, 8);
#pragma DATA_ALIGN(yk_sp_ti, 8);
#pragma DATA_ALIGN(yk_sp, 8);
#pragma DATA_ALIGN(xk_sp_l2, 8);
#pragma DATA_ALIGN(yk_sp_l2, 8);
#pragma DATA_ALIGN(xk_sp_l1d, 8);
#pragma DATA_ALIGN(yk_sp_l1d, 8);
#pragma DATA_ALIGN(a_sp_l1d, 8);
```

- Quel est le rôle de la directive #pragma DATA_SECTION ? Quel étage du processus de compilation et d'édition des liens est concerné ?

- Quel est le rôle de la directive #pragma DATA_ALIGN ? Quel étage du processus de compilation et d'édition des liens est concerné ?

- Pourquoi les tailles des vecteurs placés en L2 SRAM occupent-elles ~128Ko ? Aurait-on pu prendre des vecteurs de 256Ko ? Si oui, quelle est la limite et qu'aurions nous dû modifier dans notre projet ?

- Remplacer le fichier firtest_perf.c actuel par celui présent dans disco/c6678/sram_cache/firtest_perf .c et analyser le programme fourni réalisant des copies des vecteurs d'entrée (xk_sp) et de sortie (yk_sp) par tronçons de DDR vers L2 SRAM adressable

```
...
#if ( TEST_FIR_L2SRAM_L1DCACHE != 0 )
  else if ( memoryModel == UMA_L2SRAM_L1DCACHE ) {
       /* caches levels initializations */
       CACHE_setL2Size(CACHE_32KCACHE);
       CACHE_setL1DSize(CACHE_L1_32KCACHE);
       CACHE_setL1PSize(CACHE_L1_32KCACHE);

       start = CSL_tscRead ();

       /* copy part of input array from DDR to L2 SRAM */
       for(i=0; i<DDR_ARRAY_LENGTH; i+=L2_ARRAY_LENGTH){

              /* memory copy from DDR to L2 SRAM */
              if( i < (DDR_ARRAY_LENGTH - L2_ARRAY_LENGTH) ){
                     memcpy(xk_sp_l2, xk_sp + i, (L2_ARRAY_LENGTH + A_LENGTH - 1)*sizeof(float32_t));
              } else {
                     memcpy(xk_sp_l2, xk_sp + i, L2_ARRAY_LENGTH*sizeof(float32_t));
              }

              /* call fir algorithm for performance test */
              (*fir_fct) (xk_sp_l2, a_sp, yk_sp_l2, A_LENGTH, L2_ARRAY_LENGTH);

              /* memory copy from L2 SRAM to DDR */
              if( i < (YK_LENGTH - L2_ARRAY_LENGTH) ){
                     memcpy(output + i, yk_sp_l2, L2_ARRAY_LENGTH*sizeof(float32_t));
              } else {
                     memcpy(output + i, yk_sp_l2, (L2_ARRAY_LENGTH - A_LENGTH + 1)*sizeof(float32_t));
              }
       }

       stop = CSL_tscRead ();
       duration += stop-start;
   }
#endif

...
```

- En vous aidant de la documentation technique de CSL (Chip Support Library ou bibliothèque de fonctions pilotes des DSP C6000) présente dans le répertoire de projet /tp/doc/csl/README.md et des macros et énumérateurs présents dans le fichier d'en-tête C:\ (sur Windows) ou /opt (sous GNU\Linux) /ti/pdk_C6678_<version>/packages/ti/csl/csl_cache.h, préciser les rôles des fonctions suivantes

```
CACHE_setL2Size(CACHE_32KCACHE);
CACHE_setL1DSize(CACHE_L1_32KCACHE);
CACHE_setL1PSize(CACHE_L1_32KCACHE);
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_L2SRAM_L1DCACHE à 1

```
#define TEST_FIR_ L2SRAM_L1DCACHE  1
```

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

4.3. Préchargement des données de L2 SRAM vers L1D SRAM

- Ouvrir et analyser le programme de prototypage Matlab présent dans disco/matlab/src/main_mem_ddr_l2_l1d.m

```
for i=1 : L2_ARRAY_LENGTH : DDR_ARRAY_LENGTH

    % memcpy DDR to L2 SRAM
    if i < DDR_ARRAY_LENGTH - L2_ARRAY_LENGTH
        for k=1 : L2_ARRAY_LENGTH + A_LENGTH - 1
            xk_sp_L2(k) = xk_sp_DDR(i+k-1);
        end
    else
        for k=1 : L2_ARRAY_LENGTH
            xk_sp_L2(k) = xk_sp_DDR(i+k-1);
        end
    end

    % copy part of input array from L2 SRAM to L1D SRAM
    for j=1 : L1D_ARRAY_LENGTH : L2_ARRAY_LENGTH

        % memcpy L2 to L1D
        for k=1 : L1D_ARRAY_LENGTH + A_LENGTH - 1
            xk_sp_L1D(k) = xk_sp_L2(j+k-1);
        end

        yk_sp_L1D = fir_sp(xk_sp_L1D, a_sp, A_LENGTH, L1D_ARRAY_LENGTH);

        % memcpy L1D SRAM to L2 SRAM - coherency of output L2 array
        for k=1 : L1D_ARRAY_LENGTH
             yk_sp_L2(j+k-1) = yk_sp_L1D(k);
        end

    end

    % memcpy L2 SRAM to DDR - coherency of output DDR array
    if i < YK_LENGTH - L2_ARRAY_LENGTH
        for k=1 : L2_ARRAY_LENGTH
            yk_sp_DDR(i+k-1) = yk_sp_L2(k);
        end
    else
        for k=1 : L2_ARRAY_LENGTH - A_LENGTH + 1
            yk_sp_DDR(i+k-1) = yk_sp_L2(k);
        end
    end

end
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_L2SRAM_L1DSRAM à 1

```
#define TEST_FIR_L2SRAM_L1DSRAM    1
```

- Compléter le fichier firtest_perf.c de façon à implémenter manuellement des copies mémoires de données de DDR vers L2SRAM et de L2SRAM vers L1SRAM (cf. programme Matlab ci-dessus).

```
#if ( TEST_FIR_L2SRAM_L1DSRAM != 0 )
       else if ( memoryModel == UMA_L2SRAM_L1DSRAM ) {
             /* TODO */
       }
#endif
```

- A ce niveau là de maîtrise du modèle mémoire, quelles données restent encore présentes en cache ?

- Après validation en mode Debug, lancer une exécution en **mode Release**, compiler, tester puis reporter les résultats des tests dans le tableau d'**analyse comparative** ou **Benchmarking** présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en **mode Debug**. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

A ce stade de nos développements, nous constatons une légère perte de performance mais néanmoins un déterminisme garanti à l'exécution (maîtrise totale des données présentes dans les niveaux L2 et L1D). Juste pour information, c'était déjà partiellement le cas précédemment avec un modèle pleinement cachable dans le cadre d'un produit scalaire (données manipulées séquentiellement dans l'ordre). Néanmoins, pour grand nombre d'autres algorithmes DSP avec indexages complexes (exemple de la FFT, DCT, etc), nous aurions pu avoir de mauvaises surprises à n'utiliser que du cache.

Sans pour autant modifier notre modèle mémoire, nous avons la possibilité d'accélérer nos transferts. En effet, jusqu'à présent nous avons utilisé la fonction standard **memcpy** réalisant les transferts via le **CPU (instructions LDx/STx)**. Nous allons maintenant regarder de plus près les périphériques **DMA (Direct Memory Access)**, spécialisés dans les transferts mémoire autonomes sans passer par le CPU.


notes