Pràctica 4

Estudi del processador Cell

1. Objectius

L'objectiu d'esta pràctica és que l'alumne aprenga a programar el processador Cell i puga traure el màxim rendiment de la seua arquitectura multicomputador i SIMD. Amb esta pràctica l'alumne posarà de manifest i mesurarà l'augment de rendiment obtingut pel processament paral·lel de l'arquitectura Cell i el de les instruccions vectorials SIMD.

2. Introducció

En esta secció s'explicaran les nocions bàsiques de l'arquitectura Cell així com la seua programació utilitzant el sistema operatiu Linux instal·lat en una PlayStation 3. Una bona descripció de la programació del processador Cell es pot trobar en *Cell Broadband Engine Programming Handbook* **(CBE_Handbook_v1.1_24APR2007_pub.pub.pdf)**.

2.1 Arquitectura del processador Cell

El processador Cell està compost per un processador escalar mestre *PowerPC Processing Element* (PPE) i huit processadors independents que són els *Synergistic Processing Element* (SPE). Tots estos components estan connectats amb un bus anomenat *Element Interconnect Bus*.

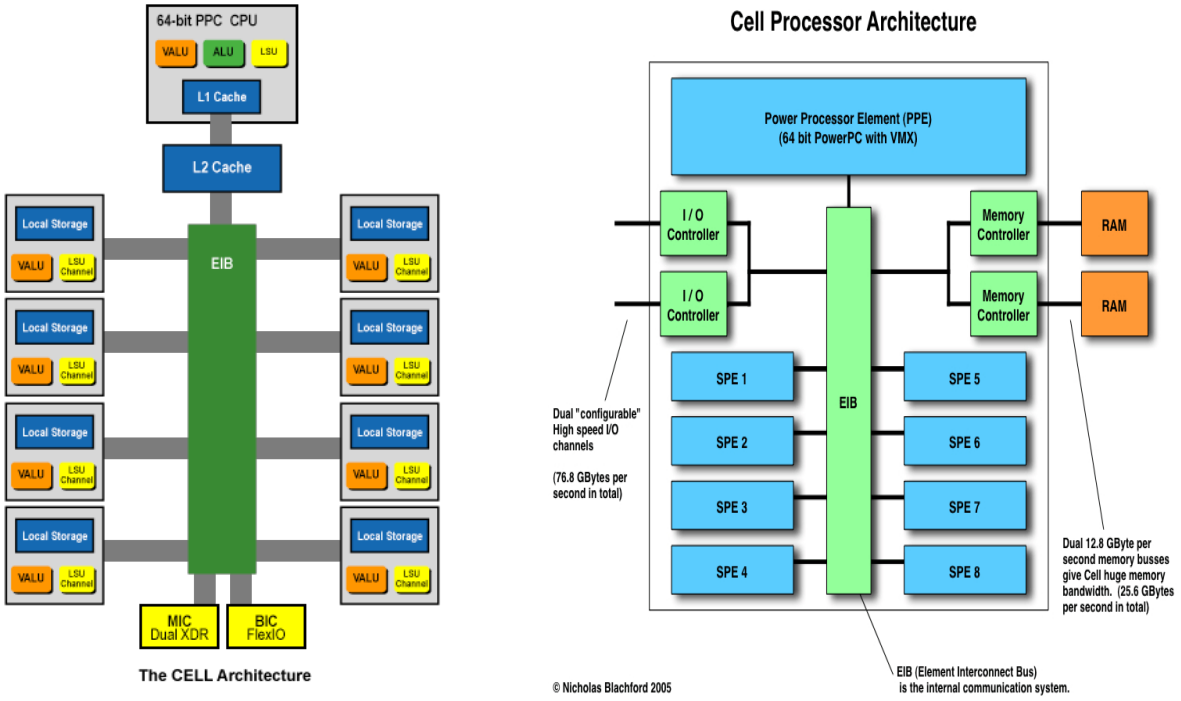

Fig. 1: Dos esquemes del processador Cell mostrant la seua arquitectura

2.1.1 El PPE: PowerPC Processing Element

Es tracta d'un processador escalar clàssic de 64 bits amb arquitectura PowerPC el que li permet executar sistemes operatius i programes estàndard per a esta família. Té instruccions RISC de 32 bits amb 64 registres de 64 bits, també posseïx una unitat de càlcul vectorial SIMD Altivec de 128 bits. El paper del PPE és sobretot el d'executar el programa principal i el de llançar els programes allotjats en els SPE.

2.1.2 Els SPE: Synergistic Processing Element

Es componen d'una unitat de càlcul vectorial *Streaming Processor Unit* (SPU) de tipus SIMD, una memòria local (*Local Store*) de 256 Kb i un *Memory Flow Controler* (MFC) per als accessos a la memòria principal. El SPE només té accés a la seua memòria local, per la qual cosa les dades han de ser transferits des de la memòria principal a la local abans de realitzar cap càlcul. Les transferències entre memòria local i principal es realitzen a través de la unitat MFC per mitjà de transaccions de tipus DMA (*Direct Memory Access*). Estes transaccions les pot iniciar el PPE o el SPE indistintament.

El SPE pot executar instruccions vectorials SIMD de 128 bits sent la grandària del vector variable depenent del tipus de dades que continga. Per exemple, els vectors són de 4 elements si contenen enters o flotants (4x4 bytes=128bits) mentres que només té 2 elements si contenen flotants dobles (2x8 bytes).

2.1.3 L'EIB: Element Interconnect Bus

És un bus intern del processador Cell que connecta tots els elements d'este. Tots els intercanvis i els accessos entre els components (SPE, PPE, memòria i interfícies de E/S) es fan per mitjà d'este bus. Cada element té una connexió de 16 bits d'escriptura i altres 16 bits de lectura amb este bus.

2.2 Programació del processador Cell

Cada element processador, tant PPE com SPE, executa el seu propi programa. Atés que el PPE i els SPE presenten arquitectures diferents, cada un té el seu propi compilador. El PPE executa normalment el programa principal i és l'encarregat de llançar els programes presents en els SPE. Per a això es disposa d'un punter global en el programa del PPE que apunta al programa del SPE (o diversos punters si cada SPE executa un programa diferent). El valor d'esta variable s'establix en el moment de la compilació si els programes del PPE i SPE es compilen junts, o durant l'execució del programa principal si és el PPE el que càrrega en memòria el programa del SPE en temps d'execució.

A l'inici s'executa el programa del PPE que és l'encarregat de crear els contextos d'execució per a cada SPE. Hi ha diversos models de programació segons el paper de cada element, el més clàssic consistix en el fet que el PPE assigna una tasca a un SPE (o a diversos) que l'executa i torna el resultat al PPE. També es pot programar en mode *streaming* de manera que un SPE fa un primer càlcul i li passa el resultat a un segon que fa un segon càlcul i així successivament. En este model cada SPE o grup de SPE efectua una acció diferent sobre les dades. També es pot usar *double buffering* el que permet solapar els temps de còpia amb el temps de càlcul.

A part de les transaccions DMA entre memòria local i principal, hi ha la possibilitat que els diferents components del processador es comuniquen entre si per mitjà de bústies de correu, de manera que hi ha una bústia d'entrada i un altre d'eixida per cada SPE i el PPE.

2.2.1 Compilació d'un programa simple

Tal com s'ha dit anteriorment s'usa un compilador diferent per al PPE i el SPE. La compilació d'un programa per al Cell necessita tres etapes: Primer cal compilar el codi del SPE amb el programa *spu-gcc*. Segon s'usa la ferramenta *ppu-embedspu* per a convertir-ho a un fitxer tipus ELF (*Executable and Linking Format*) i poder exportar l'inici del programa en una variable global perquè el PPE sàpia on comença el codi de programa. Estes dos primeres etapes s'han de repetir per cada programa diferent que executen diferents SPE; si tots els SPE executen el mateix programa (cosa habitual per un altre costat) només és necessari repetir este pas una vegada. Per fi es compila el codi del PPE amb el compilador *gcc* estàndard per exemple enllaçant-ho amb els fitxers objectes de cada programa que s'execute en el SPE.

El cas més simple i habitual és que tots els SPE executen la mateixa tasca com si d'una arquitectura SIMD es tractara (encara que no ho és). El fitxer *makefile* per a compilar el programa dels SPE (*spe.c*) i el del PPE (*ppe.c*) per a generar el programa *prog* seria:

```
OPT = -W -Wall
SPUFLAG = $(OPT)
PPUFLAG = -D_REENTRANT $(OPT) -m64

LD = -lspe2 -lpthread
```

```
prog: spe.c ppe.c
# Compilacion del codi del SPE
      spu-gcc spe.c $(SPUFLAG) -o spe
# Exportacion del codi del SPE en la variable spe_code
      ppu-embedspu spe_code spe spe.o
# Compilacion del codi del PPE
      gcc ppe.c spe.o $(PPUFLAG) $(LD) -o prog
```

En el compilador es poden afegir les opcions clàssiques que siguen necessàries com qualsevol altra compilació clàssica. En estos programes s'afigen sempre les llibreries *spe2* i *pthread* perquè la primera conté les ferramentes bàsiques de maneig dels SPE i la segona conté el que és necessari per a la creació de fils, perquè l'execució de cada programa en un SPE es va a controlar amb un fil per a cada SPE.

2.2.2 Programació

El paper del PPE és principalment el d'organitzar l'execució del programa i assignar les tasques als SPE. La primera tasca del programa del PPE és la creació dels contextos d'execució, u per cada SPE usat. La segona consistix a carregar en este context un programa que és el que serà executat sobre el SPE corresponent. L'última etapa és la d'iniciar l'execució sobre el SPE per mitjà d'una crida que es bloqueja fins al final de l'execució del SPE. No es pot triar un SPE en particular, sinó que el sistema tria el SPE que li convé. Atés que la crida al programa del SPE es bloqueja fins que acaba, és necessari usar els *pthread* per a poder continuar usant el PPE per a llançar més execucions en altres SPE o simplement que continue el programa mentres els SPE fan la seua labor. El normal és que el PPE cas els programes dels SPE i espere que acaben tots els fils per a arreplegar els resultats i continuar.

Creació d'un context d'execució

A continuació es fa una descripció ràpida de les funcions més útils per a la preparació d'un context d'execució, la descripció completa es troba en el document *SPE Runtime Management Library* ***(libspe-v2.0.pdf)***.

spe_context_ptr_t spe_context_crea't(unsigned int flags, spe_gang_context_ptr_t gang)

Esta funció crega un context d'execució i torna el seu punter o NULL en cas d'error. L'argument flags permet definir opcions com SPE_EVENT_ENABLE i altres. L'argument *gang* permet agregar este context a un *gang* (grup de SPE). En el cas més senzill i habitual simplement farem context=*spe_context_crea't(0,NULL)*

int spe_context_destroy (spe_context_ptr_t spe)

Esta funció permet destruir un context. Ha de ser usada al final de l'execució d'un context.

*int spe_program_lloeu (spe_context_ptr_t spe, spe_program_handle_t *program)*

Càrrega en el context *spe* el programa carregat en memòria en la adreça *program*. Esta adreça s'ha obtingut tal com s'explica en la secció de compilació del programa (és la variable *spe_code* del *makefile*). En el programa del PPE la definirem globalment com *extern spe_program_handle_t spe_code;*

*int spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)*

Esta funció llança l'execució del context sobre un SPE sense que podem triar un SPE en particular. La crida a esta funció es bloqueja fins que s'acaba el programa del SPE, per això és necessari crear un fil per cada SPE en el codi del PPE i cridar a esta funció des d'eixe fil.

El programa comença en el punt que indique *entry*: si el seu valor és SPE_DEFAULT_ENTRY l'execució començaré en la funció *maet*. Tant *argp* com *envp* són paràmetres que se li passen al programa SPE. L'estructura *stopinfo* permet obtindre informacions sobre el motiu del fi del programa i pot ser NULL. La crida més simple i habitual d'esta funció és *spe_context_run(ctx,&entry,0,NULL, NULL,NULL)* on *entry* serà un sencer a què li podem assignar SPE_DEFAULT_ENTRY per exemple.

Transferència de dades

Cada SPE té la seua pròpia memòria local de dades sobre la qual treballa. No té accés compartit a la memòria principal del sistema pel que les dades han de copiar-se de la memòria principal a la local abans de treballar amb ells; una vegada obtinguts els resultats estos es copien de la memòria local a la principal. Estes transaccions entre la memòria principal i la local es realitzen per mitjà d'operacions de DMA (Direct Memory Access) a través del bus EIB i poden ser llançades tant pel SPE com pel PPE.

Transferències iniciades pel SPE:

#include <spu_mfcio.h>

Este fitxer conté les definicions de les rutines utilitzades per a la transferència de dades.

mfc_tag_reserve()

mfc_tag_rellija's(unsigned int tag)

Estes funcions permeten reservar un *tag* per a una transacció i alliberar-lo quan s'haja acabat. La primera torna un sencer amb el valor del *tag* si tot ha anat bé o MFC_TAG_INVALID si s'ha produït un error.

*mfc_get(unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)*

*mfc_put(unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)*

Estes funcions són les que servixen per a llegir un bloc de la memòria principal a la local (*mfc_get*) i per a escriure un bloc de la memòria local a la principal (*mfc_put*). El bloc en memòria local comença en la adreça *lsa* i el bloc en memòria principal es troba en la adreça *ea*. El paràmetre *size* conté la grandària en bytes del bloc a transferir. El paràmetre *tag* identifica cada transacció i s'ha degut obtindre prèviament amb les funcions descrites anteriorment.

mfc_write_tag_mask(1<<tag);

mfc_read_tag_estatus_all();

Una vegada que s'han invocat les instruccions *mfc_get* i *mfc_write* hem d'assegurar-nos de que les transaccions han acabat. Per a això s'utilitzen estes dos instruccions. La primera posa la màscara per a seleccionar la transacció *tag* que volem monitoritzar, mentres que la segona s'espera a què totes les transaccions monitoritzades segons la màscara acaben.

Transferències iniciades pel PPE

El PPE també pot iniciar transferències entre la memòria principal i les locals dels SPE. A continuació s'exposen les principals:

*int spe_mfcio_put (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)*

*int spe_mfcio_get (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)*

*int spe_mfcio_tag_estatus_read(spe_context_ptr_t spe, unsigned int mask, unsigned int behavior, unsigned int *tag_estatus)*

El funcionament d'estes funcions és semblant a les seues homòlogues per als SPE. Cal fer notar no obstant això que ací el *put* i el *get* es continuen referint al SPE pel que una instrucció *put,* per exemple, còpia de la memòria local a la principal al contrari del que podria pensar-se per ser el PPE el que inicia la transacció.

El paràmetre *behavior* de la instrucció *spe_mfcio_tag_estatus_read* permet posar el valor SPE_TAG_ALL per a iniciar l'espera perquè acabe la transacció. En esta instrucció s'inclou també la màscara pel que no és necessària una instrucció addicional per a este menester.

Alineament i grandària dels Dades

A causa de la forma optimitzada en la qual es realitzen les transaccions entre la memòria principal i les locals, les dades a transferir han de complir estes dos regles:

1. La grandària del bloc a transferir ha de ser sempre múltiple de 16 bytes (4 enters).
2. L'alineament en memòria del bloc ha de ser de 16 bytes.

S'ha de parar atenció al definir la grandària de les dades i vectors a transferir afegint els bytes necessaris a les dades perquè la grandària siga sempre un múltiple de 16 bytes.

Assegurar l'alineament de les dades en memòria és més fàcil perquè el compilador disposa de la directiva *aligned* que permet alinear les dades en memòria com es desitge. Exemple:

int a[16] __attribute__(aligned(16));

El vector *a* definit anteriorment es troba alineat correctament en memòria. L'element *a[0]* està correctament alineat, però per exemple, els elements *a[1], a[2]* i *a[3]* no estan ben alineats, ja que comencen diversos bytes després del punt d'alineament. El *a[4]* no obstant això torna a estar alineat perquè *4*sizeof(int)=16* i no donaria problemes.

Instruccions SIMD

Tant el SPE com PPE disposen d'un joc d'instruccions vectorials SIMD. Estes operacions funcionen sobre vectors de 128 bits que es poden agrupar en vectors de 16 caràcters o 8 enters curts o 4 enters o 4 flotants o 2 dobles. Els vectors a operar han d'estar també alineats a 16 bytes en memòria, per a la qual cosa es pot utilitzar la mateixa directiva *aligned* ja vista anteriorment. Els detalls específics de les instruccions SIMD es troben en el manual *Language Extensions for CBEA 2.5* **(Language_Extensions_for_CBEA_2.5.pdf)** i també en *SIMD Library Specification for CBEA v10* **(SIMD_Library_Specification_for_CBEA_1.1.pdf)**.

És necessari incloure la capçalera *spu_intrinsics.h* en tots els programes de la SPE que vagen a fer ús de les instruccions SIMD.

Les instruccions SIMD vectorials utilitzen el tipus especial *vector*. Podem definir un vector de la manera següent:

vector signed int va={1,2,3,4};

També es pot definir una variable de tipus *vector* a partir d'un vector tradicional que hem pogut definir per a ser usat en altres contextos:

int b[8] __attribute__(aligned(16))={1,2,3,4,5,6,7,8}; //vector tradicional per a instruccions escalares

*vector signed int *vb=(vector signed int *)b; // vector per a instruccions SIMD*

En este cas es pot usar el vector *b* amb instruccions escalares, o el *vb* per a instruccions vectorials SIMD. En este exemple s'ha definit un vector de dos vectors: *vb[0]={1,2,3,4}* i *vb[1]={5,6,7,8}*.

Exemples d'instruccions vectorials executades en els SPE:

int a=[4] __attribute__(aligned(16))={1,2,3,4};

*vector signed int *va=(vector signed int *)a;*

vector signed int vb,vc;

*vb=spu_add(*va,1) ; // a cada element de a se li suma 1*

*vc=spu_mul(vb,*va); // multiplica els elements de vb pels de va*

vc=spu_add(vb,vc); // suma els elements de vb i vc

Altres operacions

El processador Cell permet diverses operacions addicionals a les exposades anteriorment i que permeten un millor aprofitament de l'arquitectura. Algunes d'estes operacions inclouen l'ús de bústies de correu per a la comunicació de dades entre els elements de procés, gestor d'esdeveniments i interrupcions, etc. Estes operacions estan fora d'una sessió de laboratori introductòria com esta.

3. Desenrotllament de la pràctica

En esta sessió de laboratori es compilaran diversos programes per a l'arquitectura Cell utilitzant equips PlayStation 3 amb Linux i totes les ferramentes de compilació instal·lades. Cada parella de laboratori es connectarà a una de les Playstation 3 per mitjà de *ssh*. No cal utilitzar cap de les ferramentes gràfiques de la pròpia PlayStation però si es vol llançar alguna és necessari haver-se connectat amb l'opció *–X* del *ssh*.

3.1 Programa inicial d'exemple

En el directori compartit */iilabs/AAC/ps3* es troba el directori *vector*, amb les fonts d'un programa d'exemple per al processador Cell. En este programa d'exemple se li suma un valor a tots els elements d'un vector. Per a això es trosseja el vector i es repartix cada tros a un SPE perquè el processe en paral·lel.

El fitxer *makefile* present en el directori compilarà el programa fent simplement *make*. Açò generarà l'executable *prog* que admet un paràmetre que és el número de SPEs que volem utilitzar; el mínim és 1 i el màxim 6. Este fitxer *makefile* és idèntic a l'exposat en la secció de programació més arrere.

A continuació s'exposa el codi dels fitxers que definixen el programa. Amb les explicacions donades en les seccions anteriors i els propis comentaris del codi, hauria de ser prou per a entendre el funcionament del programa.

3.1.1 Fitxer vector.h

Este fitxer conté definicions comunes als programes que es vagen a desenrotllar. En este cas es definix el nombre màxim de SPEs a utilitzar (limitada a 6), la longitud del vector i es dóna la definició de l'estructura que permet passar-li paràmetres al SPE des del PPE.

```
#define MAX_SPE 6
#define LON_VECTOR 3600  /* que siga múltiple de 120 per a alineament, més de 3600
no val perquè ocupa mes de 16k. */

/* La longitud de l'estructura ha de ser múltiple de 16 bytes, omplir amb "extres"
perquè tinga la longitud adequada. */
typedef struct {unsigned long long op1; unsigned long long cap; int lon; int
extres[3];} argument;
```

3.1.2 Fitxer ppe.c

Este fitxer conté el programa que prepara els fils i els contextos dels SPE i els repartix el vector a processar arreplegant el resultat. En esta rutina s'utilitza la funció estafes*()* per a mesurar el temps d'execució del codi i així comprovar l'augment de rendiment obtingut segons la configuració.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
#include <libspe2.h>
#include <sys/estafes.h>
#include "vector.h"

extern spe_program_handle_t spe_code;

void * thread(void * data)
{
      unsigned int entry;
      spe_context_ptr_t ctx;
argument *dades = (argument *)data;

      /* Inicializacion del context */
      ctx = spe_context_crea't(0,NULL);
      entry = SPE_DEFAULT_ENTRY;
      spe_program_lloeu(ctx,&spe_code);

      /* Ejecuta el context sobre un SPE, el fil es bloqueja fins al final de
l'execució */
      spe_context_run(ctx,&entry,0,(void *)dades, NULL,NULL);
      spe_context_destroy(ctx);
      return (void *) 0;
}


int maet(int argc, char ** argv)
{
      int i,nbspe;
      struct tms tmsaux;
      clock_t'ini,fin;
      argument datosHilo[MAX_SPE] __attribute__((aligned(16)));
      int operando1[LON_VECTOR] __attribute__((aligned(16)));
      int resultat[LON_VECTOR] __attribute__((aligned(16)));

      /*lista dels identificadors de pthreads*/
      pthread_t threads[MAX_SPE];

      if (argc < 2)
      {
            printf("Es requerix almenys un paràmetre amb el número de SPEs a
usar.\n");
            return -1;
      }

      nbspe = atoi(argv[1]);
      if (nbspe>6)
      {
            printf("Error: numere màxim de SPE disponibles = 6\n");
            return -1;
      }


/* Creación dels vectors a operar */
for (i=0;i<LON_VECTOR;i++)
        {
operando1[i]=i;
        }

/* Creación dels arguments dels fils */

      // ATENCION: LON_VECTOR ha de ser múltiple de 4, 5 i 6 (múltiple de 120)
for(i=0;i<nbspe;i++)
      {
            int tros; // Els trossos han de ser múltiples de 16 bytes (enters
múltiples de 4)
            tros=LON_VECTOR/nbspe;
            datosHilo[i].op1=(unsigned long long)&(operando1[i*tros]);
            datosHilo[i].res=(unsigned long long)&(resultat[i*tros]);
            datosHilo[i].lon=tros;
      }
```

```
ini=estafes(&tmsaux);   // Inici del temps d'execució

/* Creacion dels fils */
      for(i=0;i<nbspe;i++)
      {
            /*Iniciar un fil per cada SPE */
            if (pthread_crea't(threads+i,NULL,thread,&(datosHilo[i])) != 0)
            {
                  perror("Thread crea't.");
                  return -1;
            }
      }

      /* Esperar el fi de cada fil, la funció pthread_join espera el fi del
thread, i arreplega el valor tornat per este en el segon argument */
      for(i=0;i<nbspe;i++)
      {
            pthread_join(threads[i],NULL);
      }


      fi=estafes(&tmsaux);  // Final del temps d'execució

      /* Muestra resultats */

      // comprueba resultat
      for (i=0;i<LON_VECTOR;i++)
      {
      if (operando1[i]-1!=resultat[i])
      printf("%3d -> %3d\n",operando1[i],resultat[i]);
      }

      printf("\nTiempo: %ld\n", fi-ini);

      printf("\nFin del programa.\n");

      return 0;
}
```

3.1.3 Fitxer spe.c

En este fitxer es troba el programa que s'executa en els SPE. Primer es lligen els paràmetres enviats pel PPE on s'especifiquen l'inici del tros de vector a processar, la seua longitud i l'inici del bloc on guardar el resultat. Posteriorment es transferixen les dades del tros de vector a processar, es processen i es tornen a la memòria principal. Atés que este codi s'executa molt ràpid, a penes hi ha temps suficient per a poder realitzar mesures temporals fiables, per eixa raó s'ha afegit un bucle que repetix la mateixa operació un bon nombre de vegades i així tindre un temps d'execució raonable.

```
#include <stdio.h>
#include <spu_mfcio.h>
#include "vector.h"


int maet(unsigned long long spe_aneu, unsigned long long arg, unsigned long long
env)
{
int i,j;
uint32_t tag;
argument dada __attribute__((aligned(16)));
int op1[LON_VECTOR] __attribute__((aligned(16)));
int cap[LON_VECTOR] __attribute__((aligned(16)));


/* Trae els parametres de la mem principal (arg) al SPE (dada)*/
```

```
if((tag=mfc_tag_reserve())==MFC_TAG_INVALID) // reserva una etiqueta de transacció
 {
printf("SPE: ERROR - No es pot reservar un tag de trasaccion.\n");
return 1;
 }
mfc_get(&dada,arg,sizeof(argumento),tag,0,0);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_estatus_all();


/* Trae els parametres de la mem principal (dato.op1) al SPE (op1)*/

mfc_get(op1,dato.op1,sizeof(int)*dada.dato.lon,tag,0,0);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_estatus_all();

/* Operaciones a realitzar */

// bucle per a fer temps
for (j=0;j<200000;j++)
 {
for (i=0;i<dada.dato.lon;i++)
   {
cap[i]=op1[i]-1;
   }
 }

/* Copiar el resultat local de cap en la memòria principal a la adreça dada.cap*/

mfc_put(res,dada.cap,dato.lon*sizeof(int),tag,0,0);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_estatus_all();

mfc_tag_rellija's(tag); // allibera l'etiqueta de transacció

return 0;
}
```

3.2 Tasques a realitzar

3.2.1 Augment del rendiment amb l'ús de diversos SPE

En esta part es pretén posar de manifest i mesurar l'augment de rendiment que s'obté pel fet d'augmentar el número de SPEs entre els que es repartix la tasca a realitzar. Per a això es realitzaran diverses execucions del programa *prog* especificant un número diferent de SPE, des d'1 fins a 6.

Els resultats s'anotaran en un full de càlcul per a poder representar gràficament l'augment de rendiment en funció del número de SPEs utilitzat.

S'ha de contestar a les següents preguntes, raonant cada una d'elles:

S'acosta el resultat obtingut a què penseu que pot ser l'ideal?

Per què sí o per què no el resultat obtingut s'acosta a l'ideal?

Et pareix que amb una arquitectura de memòria compartida s'haguera obtingut un resultat millor o pitjor?

Es pot dir que esta arquitectura escala bé per al problema específic tractat?

3.2.2 Augment de rendiment amb l'augment artificial de les transaccions

En este apartat s'ha de repetir el mateix experiment de l'apartat anterior però modificant el programa del SPE (*spe.c*) perquè faça més transaccions de les realment necessàries. Per a això agafarem el bucle *for* que perd temps i ho traurem fora, de manera que englobe també a les transaccions d'escriptura i lectura amb la memòria.

Cal contestar a les següents preguntes raonant cada resposta:

S'observa alguna diferència en la gràfica de l'augment de rendiment respecte de l'anterior?

És normal esta diferència si n'hi ha?

3.2.3 Augment de rendiment amb l'ús d'instruccions SIMD

En este últim apartat haurem de substituir l'operació que es realitza per a cada element del vector per una instrucció vectorial que en este cas executarà la suma de 4 elements d'una sola vegada. Per a este experiment partirem del codi inicial del vector simple (el programa del PPE és el mateix, l'única cosa que canvia és el programa del SPE).

La instrucció de suma vectorial és *spu_add(a,b)* on *a* és un vector i *b* pot ser un vector o un escalar; si és un escalar llavors se suma eixe valor a tots els elements de *a*. No cal oblidar incloure la capçalera <*spu_intrinsics.h*> a l'inici del programa. L'altra modificació a realitzar consistix a definir dos variables tipus *vector* a partir dels vectors *op1* i *cap* que ja estan definits. Cal tindre també en compte que per cada instrucció vectorial s'executen quatre elements, per la qual cosa el bucle que recorre el vector es repetix la quarta part que el cas escalar.

Una vegada creat el programa s'executarà amb diferent nombre de SPEs i es mesurarà l'augment de rendiment. Per un costat es compararà l'augment de rendiment respecte a un processador amb SIMD per si veiem que escala de forma diferent respecte del primer experiment. Posteriorment es compararà el rendiment amb respecte d'un processador sense SIMD.

Una vegada realitzats els experiments i les gràfiques, cal contestar a les següents preguntes raonant cada resposta:

Es produïx un escalat semblant al del primer experiment?

S'acosta el resultat obtingut a què penseu que pot ser l'ideal?

Quant millora el rendiment de l'execució vectorial enfront de l'escalar del primer experiment?

Per què sí o per què no el resultat obtingut s'acosta a l'ideal o inclús ho millora?

3.2.4 Exercici addicional

Com a exercici addicional se li proposa a l'estudiant que realitze un programa per a la multiplicació de matrius utilitzant el nombre més gran possible de SPEs i instruccions vectorials amb l'objectiu d'obtindre un codi tan ràpid com siga possible. Les matrius seran quadrades de 60x60 elements.

4. Agraïments

Alguns continguts d'esta pràctica, especialment la introducció, s'han arreplegat dels treballs de Miguel Ángel Expósito Sánchez, estudiant d'Enginyeria Informàtica de l'ETSE de la Universitat de València.

5. Bibliografia

Esta bibliografia junt amb documentació addicional es pot trobar en el directori compartit */iilabs/AAC/ps3/documentació* a més de en la web del laboratori.

1. *Cell Broadband Engine Programming Handbook* **(CBE_Handbook_v1.1_24APR2007_pub.pub.pdf)**
2. *SPE Runtime Management Library* ***(libspe-v2.0.pdf)***
3. *Language Extensions for CBEA 2.*5 **(Language_Extensions_for_CBEA_2.5.pdf)**
4. *SIMD Library Specification for CBEA v1.1* **(SIMD_Library_Specification_for_CBEA_1.1.pdf)**