Práctica 4

Estudio del procesador Cell

1. Objetivos

El objetivo de esta práctica es que el alumno aprenda a programar el procesador Cell y pueda sacar el máximo rendimiento de su arquitectura multicomputador y SIMD. Con esta práctica el alumno pondrá de manifiesto y medirá el aumento de rendimiento obtenido por el procesamiento paralelo de la arquitectura Cell y el de las instrucciones vectoriales SIMD.

2. Introducción

En esta sección se explicarán las nociones básicas de la arquitectura Cell así como su programación utilizando el sistema operativo Linux instalado en una PlayStation 3. Una buena descripción de la programación del procesador Cell se puede encontrar en *Cell Broadband Engine Programming Handbook* **(CBE_Handbook_v1.1_24APR2007_pub.pdf**).

2.1 Arquitectura del procesador Cell

El procesador Cell está compuesto por un procesador escalar maestro *PowerPC Processing Element* (PPE) y ocho procesadores independientes que son los *Synergistic Processing Element* (SPE). Todos estos componentes están conectados con un bus llamado *Element Interconnect Bus*.

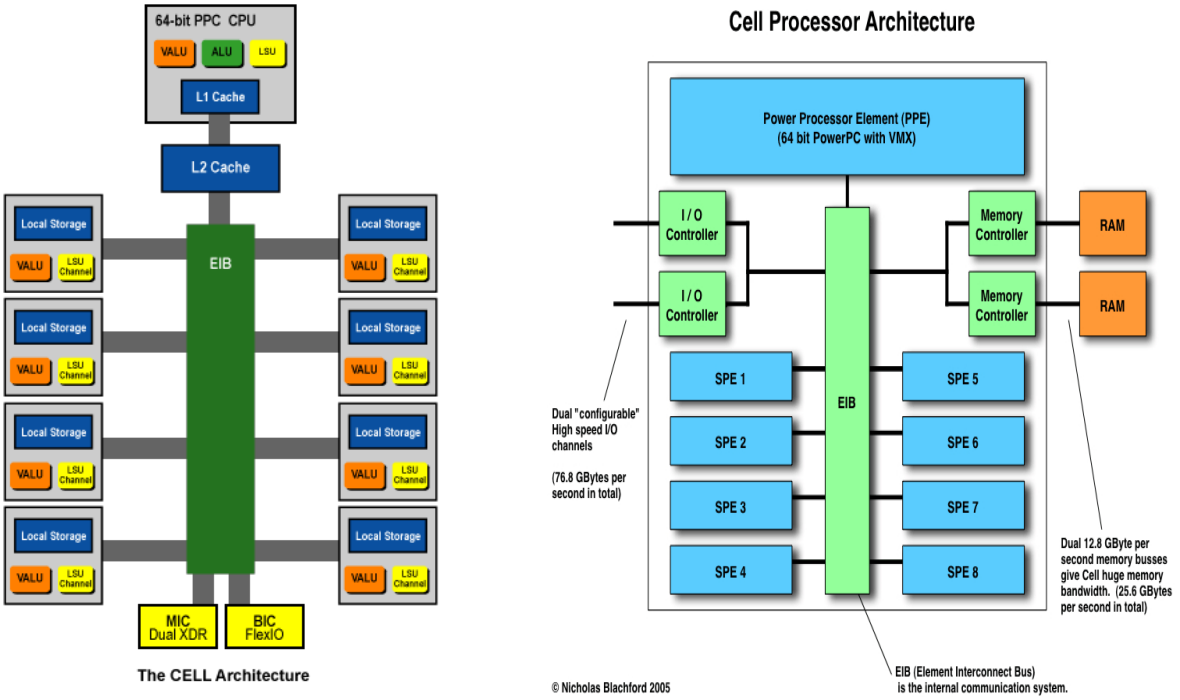

Fig. 1: Dos esquemas del procesador Cell mostrando su arquitectura

2.1.1 El PPE: PowerPC Processing Element

Se trata de un procesador escalar clásico de 64 bits con arquitectura PowerPC lo que le permite ejecutar sistemas operativos y programas estándar para esta familia. Tiene instrucciones RISC de 32 bits con 64 registros de 64 bits, también posee una unidad de cálculo vectorial SIMD Altivec de 128 bits. El papel del PPE es sobre todo el de ejecutar el programa principal y el de lanzar los programas alojados en los SPE.

2.1.2 Los SPE: Synergistic Processing Element

Se componen de una unidad de cálculo vectorial *Streaming Processor Unit* (SPU) de tipo SIMD, una memoria local (*Local Store*) de 256 Kb y un *Memory Flow Controler* (MFC) para los accesos a la memoria principal. El SPE sólo tiene acceso a su memoria local, por lo que los datos deben ser transferidos desde la memoria principal a la local antes de realizar ningún cálculo. Las transferencias entre memoria local y principal se realizan a través de la unidad MFC mediante transacciones de tipo DMA (*Direct Memory Access*). Estas transacciones las puede iniciar el PPE o el SPE indistintamente.

El SPE puede ejecutar instrucciones vectoriales SIMD de 128 bits siendo el tamaño del vector variable dependiendo del tipo de datos que contenga. Por ejemplo, los vectores son de 4 elementos si contienen enteros o flotantes (4x4 bytes=128bits) mientras que sólo tiene 2 elementos si contienen flotantes dobles (2x8 bytes).

2.1.3 El EIB: Element Interconnect Bus

Es un bus interno del procesador Cell que conecta todos los elementos de este. Todos los intercambios y los accesos entre los componentes (SPE, PPE, memoria e interfaces de E/S) se hacen mediante este bus. Cada elemento tiene una conexión de 16 bits de escritura y otros 16 bits de lectura con este bus.

2.2 Programación del procesador Cell

Cada elemento procesador, tanto PPE como SPE, ejecuta su propio programa. Dado que el PPE y los SPE presentan arquitecturas diferentes, cada uno tiene su propio compilador. El PPE ejecuta normalmente el programa principal y es el encargado de lanzar los programas presentes en los SPE. Para ello se dispone de un puntero global en el programa del PPE que apunta al programa del SPE (o varios punteros si cada SPE ejecuta un programa diferente). El valor de esta variable se establece en el momento de la compilación si los programas del PPE y SPE se compilan juntos, o durante la ejecución del programa principal si es el PPE el que carga en memoria el programa del SPE en tiempo de ejecución.

Al inicio se ejecuta el programa del PPE que es el encargado de crear los contextos de ejecución para cada SPE. Existen varios modelos de programación según el papel de cada elemento, el más clásico consiste en que el PPE asigna una tarea a un SPE (o a varios) que la ejecuta y devuelve el resultado al PPE. También se puede programar en modo *streaming* de manera que un SPE hace un primer cálculo y le pasa el resultado a un segundo que hace un segundo cálculo y así sucesivamente. En este modelo cada SPE o grupo de SPE efectúa una acción diferente sobre los datos. También se puede usar *double buffering* lo que permite solapar los tiempos de copia con el tiempo de cálculo.

Aparte de las transacciones DMA entre memoria local y principal, existe la posibilidad de que los diferentes componentes del procesador se comuniquen entre sí mediante buzones de correo, de manera que existe un buzón de entrada y otro de salida por cada SPE y el PPE.

2.2.1 Compilación de un programa simple

Tal como se ha dicho anteriormente se usa un compilador diferente para el PPE y el SPE. La compilación de un programa para el Cell necesita tres etapas: Primero hay que compilar el código del SPE con el programa *spu-gcc*. Segundo se usa la herramienta *ppu-embedspu* para convertirlo a un fichero tipo ELF (*Executable and Linking Format*) y poder exportar el inicio del programa en una variable global para que el PPE sepa dónde comienza el código de programa. Estas dos primeras etapas se deben repetir por cada programa diferente que ejecuten diferentes SPE; si todos los SPE ejecutan el mismo programa (cosa habitual por otro lado) sólo es necesario repetir este paso una vez. Por fin se compila el código del PPE con el compilador *gcc* estándar por ejemplo enlazándolo con los ficheros objetos de cada programa que se ejecute en el SPE.

El caso más simple y habitual es que todos los SPE ejecuten la misma tarea como si de una arquitectura SIMD se tratara (aunque no lo es). El fichero *makefile* para compilar el programa de los SPE (*spe.c*) y el del PPE (*ppe.c*) para generar el programa *prog* sería:

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

LD = -lspe2 -lpthread
```

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

En el compilador se pueden añadir las opciones clásicas que sean necesarias como cualquier otra compilación clásica. En estos programas se añaden siempre las librerías *spe2* y *pthread* pues la primera contiene las herramientas básicas de manejo de los SPE y la segunda contiene lo necesario para la creación de hilos, pues la ejecución de cada programa en un SPE se va a controlar con un hilo para cada SPE.

2.2.2 Programación

El papel del PPE es principalmente el de organizar la ejecución del programa y asignar las tareas a los SPE. La primera tarea del programa del PPE es la creación de los contextos de ejecución, uno por cada SPE usado. La segunda consiste en cargar en este contexto un programa que es el que va a ser ejecutado sobre el SPE correspondiente. La última etapa es la de iniciar la ejecución sobre el SPE mediante una llamada que se bloquea hasta el fin de la ejecución del SPE. No se puede elegir un SPE en particular, sino que el sistema elige el SPE que le conviene. Dado que la llamada al programa del SPE se bloquea hasta que termina, es necesario usar los *pthread* para poder continuar usando el PPE para lanzar más ejecuciones en otros SPE o simplemente que continúe el programa mientras los SPE hacen su labor. Lo normal es que el PPE lance los programas de los SPE y espere a que terminen todos los hilos para recoger los resultados y continuar.

Creación de un contexto de ejecución

A continuación se hace una descripción rápida de las funciones más útiles para la preparación de un contexto de ejecución, la descripción completa se encuentra en el documento *SPE Runtime Management Library* ***(libspe-v2.0.pdf)***.

spe_context_ptr_t spe_context_create(unsigned int flags, spe_gang_context_ptr_t gang)

Esta función crea un contexto de ejecución y devuelve su puntero o NULL en caso de error. El argumento flags permite definir opciones como SPE_EVENT_ENABLE y otras. El argumento *gang* permite agregar este contexto a un *gang* (grupo de SPE). En el caso más sencillo y habitual simplemente haremos *contexto=spe_context_create(0,NULL)*

int spe_context_destroy (spe_context_ptr_t spe)

Esta función permite destruir un contexto. Tiene que ser usada al final de la ejecución de un contexto.

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

Carga en el contexto *spe* el programa cargado en memoria en la dirección *program*. Esta dirección se ha obtenido tal como se explica en la sección de compilación del programa (es la variable *spe_code* del *makefile*). En el programa del PPE la definiremos globalmente como *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 función lanza la ejecución del contexto sobre un SPE sin que podamos elegir un SPE en particular. La llamada a esta función se bloquea hasta que se termina el programa del SPE, por eso es necesario crear un hilo por cada SPE en el código del PPE y llamar a esta función desde ese hilo.

El programa empieza en el punto que indique *entry*: si su valor es SPE_DEFAULT_ENTRY la ejecución empezaré en la función *main*. Tanto *argp* como *envp* son parámetros que se le pasan al programa SPE. La

estructura *stopinfo* permite obtener informaciones sobre el motivo del fin del programa y puede ser NULL. La llamada más simple y habitual de esta función es *spe_context_run(ctx,&entry,0,NULL, NULL,NULL)* donde *entry* será un entero al que le podemos asignar SPE_DEFAULT_ENTRY por ejemplo.

Transferencia de datos

Cada SPE tiene su propia memoria local de datos sobre la que trabaja. No tiene acceso compartido a la memoria principal del sistema por lo que los datos deben copiarse de la memoria principal a la local antes de trabajar con ellos; una vez obtenidos los resultados éstos se copian de la memoria local a la principal. Estas transacciones entre la memoria principal y la local se realizan mediante operaciones de DMA (Direct Memory Access) a través del bus EIB y pueden ser lanzadas tanto por el SPE como por el PPE.

Transferencias iniciadas por el SPE:

#include <spu_mfcio.h>

Este fichero contiene las definiciones de las rutinas utilizadas para la transferencia de datos.

mfc_tag_reserve()

mfc_tag_release(unsigned int tag)

Estas funciones permiten reservar un *tag* para una transacción y liberarlo cuando se haya terminado. La primera devuelve un entero con el valor del *tag* si todo ha ido bien o MFC_TAG_INVALID si se ha producido 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)*

Estas funciones son las que sirven para leer un bloque de la memoria principal a la local (*mfc_get*) y para escribir un bloque de la memoria local a la principal (*mfc_put*). El bloque en memoria local empieza en la dirección *lsa* y el bloque en memoria principal se encuentra en la dirección *ea*. El parámetro *size* contiene el tamaño en bytes del bloque a transferir. El parámetro *tag* identifica cada transacción y se ha debido obtener previamente con las funciones descritas anteriormente.

mfc_write_tag_mask(1<<tag);

mfc_read_tag_status_all();

Una vez que se han invocado las instrucciones *mfc_get* y *mfc_write* debemos asegurarnos de que las transacciones han terminado. Para ello se utilizan estas dos instrucciones. La primera pone la máscara para seleccionar la transacción *tag* que queremos monitorizar, mientras que la segunda se espera a que todas las transacciones monitorizadas según la máscara terminen.

Transferencias iniciadas por el PPE

El PPE también puede iniciar transferencias entre la memoria principal y las locales de los SPE. A continuación se exponen las principales:

*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_status_read(spe_context_ptr_t spe, unsigned int mask, unsigned int behavior, unsigned int *tag_status)*

El funcionamiento de estas funciones es similar a sus homólogas para los SPE. Hay que hacer notar sin embargo que aquí el *put* y el *get* se siguen refiriendo al SPE por lo que una instrucción *put*, por ejemplo, copia de la memoria local a la principal al contrario de lo que podría pensarse por ser el PPE el que inicia la transacción.

El parámetro *behavior* de la instrucción *spe_mfcio_tag_status_read* permite poner el valor SPE_TAG_ALL para iniciar la espera para que acabe la transacción. En esta instrucción se incluye también la máscara por lo que no es necesaria una instrucción adicional para este menester.

Alineamiento y tamaño de los Datos

Debido a la forma optimizada en la cual se realizan las transacciones entre la memoria principal y las locales, los datos a transferir deben cumplir estas dos reglas:

1. El tamaño del bloque a transferir debe ser siempre múltiplo de 16 bytes (4 enteros).
2. El alineamiento en memoria del bloque debe ser de 16 bytes.

Se debe tener cuidado al definir el tamaño de los datos y vectores a transferir añadiendo los bytes necesarios a los datos para que el tamaño sea siempre un múltiplo de 16 bytes.

Asegurar el alineamiento de los datos en memoria es más fácil pues el compilador dispone de la directiva *aligned* que permite alinear los datos en memoria como se desee. Ejemplo:

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

El vector *a* definido anteriormente se encuentra alineado correctamente en memoria. El elemento *a[0]* está correctamente alineado, pero por ejemplo, los elementos *a[1], a[2]* y *a[3]* no están bien alineados, puesto que empiezan varios bytes después del punto de alineamiento. El *a[4]* sin embargo vuelve a estar alineado pues *4*sizeof(int)=16* y no daría problemas.

Instrucciones SIMD

Tanto el SPE como PPE disponen de un juego de instrucciones vectoriales SIMD. Estas operaciones funcionan sobre vectores de 128 bits que se pueden agrupar en vectores de 16 caracteres o 8 enteros cortos o 4 enteros o 4 flotantes o 2 dobles. Los vectores a operar deben estar también alineados a 16 bytes en memoria, para lo cual se puede utilizar la misma directiva *aligned* ya vista anteriormente. Los detalles específicos de las instrucciones SIMD se encuentran en el manual *Language Extensions for CBEA 2.5* **(Language_Extensions_for_CBEA_2.5.pdf)** y también en *SIMD Library Specification for CBEA v10* **(SIMD_Library_Specification_for_CBEA_1.1.pdf)**.

Es necesario incluir la cabecera *spu_intrinsics.h* en todos los programas de la SPE que vayan a hacer uso de las instrucciones SIMD.

Las instrucciones SIMD vectoriales utilizan el tipo especial *vector*. Podemos definir un vector de la siguiente manera:

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

También se puede definir una variable de tipo *vector* a partir de un vector tradicional que hemos podido definir para ser usado en otros contextos:

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

*vector signed int *vb=(vector signed int *)b; // vector para instrucciones SIMD*

En este caso se puede usar el vector *b* con instrucciones escalares, o el *vb* para instrucciones vectoriales SIMD. En este ejemplo se ha definido un vector de dos vectores: *vb[0]={1,2,3,4}* y *vb[1]={5,6,7,8}*.

Ejemplos de instrucciones vectoriales ejecutadas en los 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 elemento de a se le suma 1*

*vc=spu_mul(vb,*va); // multiplica los elementos de vb por los de va*

vc=spu_add(vb,vc); // suma los elementos de vb y vc

Otras operaciones

El procesador Cell permite varias operaciones adicionales a las expuestas anteriormente y que permiten un mejor aprovechamiento de la arquitectura. Algunas de estas operaciones incluyen el uso de buzones de correo para la comunicación de datos entre los elementos de proceso, gestor de eventos e interrupciones, etc. Estas operaciones están fuera de una sesión de laboratorio introductoria como esta.

3. Desarrollo de la práctica

En esta sesión de laboratorio se compilarán varios programas para la arquitectura Cell utilizando equipos PlayStation 3 con Linux y todas las herramientas de compilación instaladas. Cada pareja de laboratorio se conectará a una de las Playstation 3 mediante *ssh*. No es necesario utilizar ninguna de las herramientas gráficas de la propia PlayStation pero si se quiere lanzar alguna es necesario haberse conectado con la opción *–X* del *ssh*.

3.1 Programa inicial de ejemplo

En el directorio compartido */ii/labs/AAC/ps3* se encuentra el directorio *vector*, con las fuentes de un programa de ejemplo para el procesador Cell. En este programa de ejemplo se le suma un valor a todos los elementos de un vector. Para ello se trocea el vector y se reparte cada pedazo a un SPE para que lo procese en paralelo.

El fichero *makefile* presente en el directorio compilará el programa haciendo simplemente *make*. Esto generará el ejecutable *prog* que admite un parámetro que es el número de SPEs que queremos utilizar; el mínimo es 1 y el máximo 6. Este fichero *makefile* es idéntico al expuesto en la sección de programación más atrás.

A continuación se expone el código de los ficheros que definen el programa. Con las explicaciones dadas en las secciones anteriores y los propios comentarios del código, debería ser suficiente para entender el funcionamiento del programa.

3.1.1 Fichero vector.h

Este fichero contiene definiciones comunes a los programas que se vayan a desarrollar. En este caso se define el número máximo de SPEs a utilizar (limitada a 6), la longitud del vector y se da la definición de la estructura que permite pasarle parámetros al SPE desde el PPE.

```
1  #define MAX_SPE 6
2  #define LON_VECTOR 3600  /* que sea multiplo de 120 para alineamiento, más de 3600
3  no vale pues ocupa mas de 16k. */
4
5  /* La longitud de la estructura debe ser multiplo de 16 bytes, rellenar con "extras"
6  para que tenga la longitud adecuada. */
7  typedef struct {unsigned long long op1; unsigned long long res; int lon; int
8  extras[3];} argumento;
9
```

3.1.2 Fichero ppe.c

Este fichero contiene el programa que prepara los hilos y los contextos de los SPE y les reparte el vector a procesar recogiendo el resultado. En esta rutina se utiliza la función *times()* para medir el tiempo de ejecución del código y así comprobar el aumento de rendimiento obtenido según la configuración.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <libspe2.h>
5  #include <sys/times.h>
6  #include "vector.h"
7
8  extern spe_program_handle_t spe_code;
9
10 void * thread(void * data)
11 {
12        unsigned int entry;
13        spe_context_ptr_t ctx;
14        argumento *datos = (argumento *)data;
15
16        /* Inicializacion del contexto */
17        ctx = spe_context_create(0,NULL);
18        entry = SPE_DEFAULT_ENTRY;
19        spe_program_load(ctx,&spe_code);
20
21        /* Ejecuta el contexto sobre un SPE, el hilo se bloquea hasta el fin de la
22 ejecucion */
23        spe_context_run(ctx,&entry,0,(void *)datos, NULL,NULL);
24        spe_context_destroy(ctx);
25        return (void *) 0;
26 }
27
28
29 int main(int argc, char ** argv)
30 {
31        int i,nbspe;
32        struct tms tmsaux;
33        clock_t ini,fin;
34        argumento datosHilo[MAX_SPE] __attribute__((aligned(16)));
35        int operando1[LON_VECTOR] __attribute__((aligned(16)));
36        int resultado[LON_VECTOR] __attribute__((aligned(16)));
37
38        /*lista de los identificadores de pthreads*/
39        pthread_t threads[MAX_SPE];
40
41        if (argc < 2)
42        {
43              printf("Se requiere al menos un parámetro con el número de SPEs a
44 usar.\n");
45              return -1;
46        }
47
48        nbspe = atoi(argv[1]);
49        if (nbspe>6)
50        {
51              printf("Error: numero maximo de SPE disponibles = 6\n");
52              return -1;
53        }
54
55
56         /* Creación de los vectores a operar */
57         for (i=0;i<LON_VECTOR;i++)
58         {
59              operando1[i]=i;
60         }
61
62         /* Creación de los argumentos de los hilos */
63
64        // ATENCION: LON_VECTOR debe ser múltiplo de 4, 5 y 6 (multiplo de 120)
65        for(i=0;i<nbspe;i++)
66        {
67              int trozo; // Los trozos deben ser múltiplos de 16 bytes (enteros
68 multiplos de 4)
69              trozo=LON_VECTOR/nbspe;
70              datosHilo[i].op1=(unsigned long long)&(operando1[i*trozo]);
```

```
71              datosHilo[i].res=(unsigned long long)&(resultado[i*trozo]);
72              datosHilo[i].lon=trozo;
73      }
74
75
76       ini=times(&tmsaux);   // Inicio del tiempo de ejecucion
77
78       /* Creacion de los hilos */
79      for(i=0;i<nbspe;i++)
80      {
81              /*Iniciar un hilo por cada SPE */
82              if (pthread_create(threads+i,NULL,thread,&(datosHilo[i])) != 0)
83              {
84                      perror("Thread create.");
85                      return -1;
86              }
87      }
88
89      /* Esperar el fin de cada hilo, la funcion pthread_join espera el fin del
90  thread, y recoge el valor devuelto por este en el segundo argumento */
91      for(i=0;i<nbspe;i++)
92      {
93              pthread_join(threads[i],NULL);
94      }
95
96
97      fin=times(&tmsaux);  // Final del tiempo de ejecución
98
99      /* Muestra resultados */
100
101     // comprueba resultado
102     for (i=0;i<LON_VECTOR;i++)
103     {
104       if (operando1[i]-1!=resultado[i])
105         printf("%3d -> %3d\n",operando1[i],resultado[i]);
106     }
107
108     printf("\nTiempo: %ld\n", fin-ini);
109
110     printf("\nFin del programa.\n");
111
112     return 0;
113 }
```

3.1.3 Fichero spe.c

En este fichero se encuentra el programa que se ejecuta en los SPE. Primero se leen los parámetros enviados por el PPE donde se especifican el inicio del trozo de vector a procesar, su longitud y el inicio del bloque donde guardar el resultado. Posteriormente se transfieren los datos del trozo de vector a procesar, se procesan y se devuelven a la memoria principal. Dado que este código se ejecuta muy rápido, apenas hay tiempo suficiente para poder realizar medidas temporales fiables, por esa razón se ha añadido un bucle que repite la misma operación un buen número de veces y así tener un tiempo de ejecución razonable.

```
1  #include <stdio.h>
2  #include <spu_mfcio.h>
3  #include "vector.h"
4
5
6  int main(unsigned long long spe_id, unsigned long long arg, unsigned long long env)
7  {
8  int i,j;
9  uint32_t tag;
10 argumento dato __attribute__((aligned(16)));
11 int op1[LON_VECTOR] __attribute__((aligned(16)));
12 int res[LON_VECTOR] __attribute__((aligned(16)));
13
14
```

```
 /* Trae los parametros de la mem principal (arg) al SPE (dato)*/

 if((tag=mfc_tag_reserve())==MFC_TAG_INVALID) // reserva una etiqueta de transaccion
 {
   printf("SPE: ERROR - No se puede reservar un tag de trasaccion.\n");
   return 1;
 }
 mfc_get(&dato,arg,sizeof(argumento),tag,0,0);
 mfc_write_tag_mask(1<<tag);
 mfc_read_tag_status_all();


 /* Trae los parametros de la mem principal (dato.op1) al SPE (op1)*/

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

 /* Operaciones a realizar */

 // bucle para hacer tiempo
 for (j=0;j<200000;j++)
 {
   for (i=0;i<dato.lon;i++)
   {
      res[i]=op1[i]-1;
   }
 }

 /* Copiar el resultado local de res en la memoria principal a la direccion
dato.res*/

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

 mfc_tag_release(tag); // libera la etiqueta de transaccion

 return 0;
}
```

3.2 Tareas a realizar

3.2.1 Aumento del rendimiento con el uso de varios SPE

En esta parte se pretende poner de manifiesto y medir el aumento de rendimiento que se obtiene por el hecho de aumentar el número de SPEs entre los que se reparte la tarea a realizar. Para ello se realizarán varias ejecuciones del programa *prog* especificando un número diferente de SPE, desde 1 hasta 6.

Los resultados se anotarán en una hoja de cálculo para poder representar gráficamente el aumento de rendimiento en función del número de SPEs utilizado.

Se debe contestar a las siguientes preguntas, razonando cada una de ellas:

¿Se acerca el resultado obtenido al que pensáis que puede ser el ideal?

¿Por qué sí o por qué no el resultado obtenido se acerca al ideal?

¿Te parece que con una arquitectura de memoria compartida se hubiera obtenido un resultado mejor o peor?

¿Se puede decir que esta arquitectura escala bien para el problema específico tratado?

3.2.2 Aumento de rendimiento con el aumento artificial de las transacciones

En este apartado se debe repetir el mismo experimento del apartado anterior pero modificando el programa del SPE (*spe.c*) para que haga más transacciones de las realmente necesarias. Para ello cogeremos el bucle *for* que

pierde tiempo y lo sacaremos fuera, de manera que englobe también a las transacciones de escritura y lectura con la memoria.

Hay que contestar a las siguientes preguntas razonando cada respuesta:

¿Se observa alguna diferencia en la gráfica del aumento de rendimiento respecto del anterior?

¿Es normal esta diferencia si la hay?

3.2.3 Aumento de rendimiento con el uso de instrucciones SIMD

En este último apartado deberemos sustituir la operación que se realiza para cada elemento del vector por una instrucción vectorial que en este caso ejecutará la suma de 4 elementos de una sola vez. Para este experimento partiremos del código inicial del vector simple (el programa del PPE es el mismo, lo único que cambia es el programa del SPE).

La instrucción de suma vectorial es *spu_add(a,b)* donde *a* es un vector y *b* puede ser un vector o un escalar; si es un escalar entonces se suma ese valor a todos los elementos de *a*. No hay que olvidar incluir la cabecera *<spu_intrinsics.h>* al inicio del programa. La otra modificación a realizar consiste en definir dos variables tipo *vector* a partir de los vectores *op1* y *res* que ya están definidos. Hay que tener también en cuenta que por cada instrucción vectorial se ejecutan cuatro elementos, por lo que el bucle que recorre el vector se repite la cuarta parte que el caso escalar.

Una vez creado el programa se ejecutará con diferente número de SPEs y se medirá el aumento de rendimiento. Por un lado se comparará el aumento de rendimiento con respecto a un procesador con SIMD por si vemos que escala de forma diferente respecto del primer experimento. Posteriormente se comparará el rendimiento con respecto de un procesador sin SIMD.

Una vez realizados los experimentos y las gráficas, hay que contestar a las siguientes preguntas razonando cada respuesta:

¿Se produce un escalado similar al del primer experimento?

¿Se acerca el resultado obtenido al que pensáis que puede ser el ideal?

¿Cuánto mejora el rendimiento de la ejecución vectorial frente a la escalar del primer experimento?

¿Por qué sí o por qué no el resultado obtenido se acerca al ideal o incluso lo mejora?

3.2.4 Ejercicio adicional

Como ejercicio adicional se le propone al estudiante que realice un programa para la multiplicación de matrices utilizando el mayor número posible de SPEs e instrucciones vectoriales con el objetivo de obtener un código lo más rápido posible. Las matrices serán cuadradas de 60x60 elementos.

4. Agradecimientos

Algunos contenidos de esta práctica, especialmente la introducción, se han recogido de los trabajos de Miguel Ángel Expósito Sánchez, estudiante de Ingeniería Informática de la ETSE de la Universitat de València.

5. Bibliografía

Esta bibliografía junto con documentación adicional se puede encontrar en el directorio compartido */iilabs/AAC/ps3/documentacion* además de en la web del laboratorio.

1. *Cell Broadband Engine Programming Handbook* **(CBE_Handbook_v1.1_24APR2007_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)**