Práctica 2

Multiplicación de matrices con Cell

1. Objetivos

El objetivo de esta práctica es que el alumno aprenda a utilizar las características paralelas y vectoriales del procesador Cell para realizar multiplicación de matrices.

2. Introducción

Suponiendo una multiplicación de matrices RES=AxB, Se pueden aprovechar las instrucciones vectoriales SIMD multiplicando cada elemento de la primera fila de la matriz A por cada fila de B acumulando el resultado en un vector que al finalizar contendrá la primera fila del resultado RES. Este proceso se repite para el resto de filas de A, obteniéndose el resto de filas de RES. Esto es una forma conveniente pues todas las operaciones que se realizan son vectoriales (en concreto se hace una multiplicación escalar-vector seguida de una suma de vectores).

La forma de multiplicación anterior tiene además la ventaja de ser fácilmente paralelizable por filas, de manera que cada procesador se puede encargar de unas filas del resultado, de manera que le tendremos que pasar las mismas filas de A que de resultado, aunque se tendrá que pasar toda la matriz B a todos los procesadores.

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 */ii1abs/AAC/ps3* se encuentra el directorio *matriz*, con las fuentes de un programa de ejemplo para el procesador Cell. En este programa de ejemplo se le pasan a un spe las matrices A y B para que las multiplique por el método tradicional de filas por columnas, por lo que habrá que modificarlo para paralelizarlo y vectorizarlo.

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  // 60x60=3600 debe ser multiplo de 120 para alineamiento, m?s de 3600 no vale pu
3  es ocuparia mas de 16k.
4  #define LON_FIL 60
5  #define LON_COL 60
6
7  /* La longitud de la estructura debe ser multiplo de 16 bytes, rellenar con "ext
8  ras" para que tenga la longitud adecuada. */
```

```
typedef struct {
 unsigned long long op1; // 8
 unsigned long long op2; // 8
 unsigned long long res; // 8
 int lon; // 4
 int extras[1]; // 1x4=4
} argumento;
```

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <libspe2.h>
#include <sys/times.h>
#include "vector.h"

extern spe_program_handle_t spe_code;

void * thread(void * data)
{
        unsigned int entry;
        spe_context_ptr_t ctx;
        argumento *datos = (argumento *)data;

        /* Inicializacion del contexto */
        ctx = spe_context_create(0,NULL);
        entry = SPE_DEFAULT_ENTRY;
        spe_program_load(ctx,&spe_code);

        /* Ejecuta el contexto sobre un SPE, el hilo se bloquea hasta el fin de la
ejecucion */
        spe_context_run(ctx,&entry,0,(void *)datos, NULL,NULL);
        spe_context_destroy(ctx);
        return (void *) 0;
}


int main(int argc, char ** argv)
{
        int i,j, k, nbspe;
        struct tms tmsaux;
        clock_t ini,fin;
        argumento datosHilo[MAX_SPE] __attribute__((aligned(16)));
        float a[LON_FIL*LON_COL] __attribute__((aligned(16)));
        float b[LON_FIL*LON_COL] __attribute__((aligned(16)));
        float res[LON_FIL*LON_COL] __attribute__((aligned(16)));
        float valor;

        /*lista de los identificadores de pthreads*/
        pthread_t threads[MAX_SPE];

        if (argc < 2)
        {
                printf("Se requiere al menos un par?metro con el n?mero de SPEs a
usar.\n");
                return -1;
        }

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

```
        }


        /* Creaci?n de los vectores a operar */
        for (i=0;i<LON_FIL;i++)
        {
          for (j=0;j<LON_COL;j++)
          {
             a[i*LON_COL+j]=i+3*j;
             b[i*LON_COL+j]=2*i+j;
             res[i*LON_COL+j]=0;
          }
        }

        /* Creaci?n de los argumentos de los hilos */

        // ATENCION: LON_VECTOR debe ser m?ltiplo de 4, 5 y 6 (multiplo de 60)
        // de momento usa un spe
        //for(i=0;i<nbspe;i++)
        for(i=0;i<1;i++)
        {
                int trozo; // Los trozos deben ser m?ltiplos de 16 bytes (enteros
multiplos de 4)
                trozo=LON_FIL; // contiene las filas a procesar, todas...
                // cada spe hace trozo filas de la matriz
                datosHilo[i].op1=(unsigned long long)&(a[0]);
                datosHilo[i].op2=(unsigned long long)b; // toda a todos
                datosHilo[i].res=(unsigned long long)&(res[0]);
                datosHilo[i].lon=trozo;
        }


        ini=times(&tmsaux);   // Inicio del tiempo de ejecucion

        /* Creacion de los hilos */
        //for(i=0;i<nbspe;i++)
        for(i=0;i<1;i++)
        {
                /*Iniciar un hilo por cada SPE */
                if (pthread_create(threads+i,NULL,thread,&(datosHilo[i])) != 0)
                {
                        perror("Thread create.");
                        return -1;
                }
        }

        /* Esperar el fin de cada hilo, la funcion pthread_join espera el fin del
thread, y recoge el valor devuelto por este en el segundo argumento */
        //for(i=0;i<nbspe;i++)
        for(i=0;i<1;i++)
        {
                pthread_join(threads[i],NULL);
        }


        fin=times(&tmsaux);  // Final del tiempo de ejecuci?n

        /* Muestra resultados */

        // comprueba resultado
        //for (i=0;i<LON_VECTOR;i++)
        //{
        //  if (operando1[i]-1!=resultado[i])
        //    printf("%3d -> %3d\n",operando1[i],resultado[i]);
        //}
        for (i=0;i<LON_FIL;i++)
        {
          for (j=0;j<LON_COL;j++)
          {
             valor=0;
             for (k=0;k<LON_FIL; k++) valor=valor+a[i*LON_COL+k]*b[k*LON_COL+j];
             if ((1) && (valor!=res[i*LON_COL+j]))
```

```
                printf("Error     en     (%3d,%3d)          %7.1fx%7.1f=%7.1f<>%7.1f
\n",i,j,a[i*LON_COL+j],b[i*LON_COL+j],valor,res[i*LON_COL+j]);
             }
         }

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

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

         return 0;
}
```

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.

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


int main(unsigned long long spe_id, unsigned long long arg, unsigned long long env)
{
int i,j,k,n;
uint32_t tag;
argumento dato __attribute__((aligned(16)));
float op1[LON_FIL*LON_COL] __attribute__((aligned(16)));
float op2[LON_FIL*LON_COL] __attribute__((aligned(16)));
float res[LON_FIL*LON_COL] __attribute__((aligned(16)));

float valor;

 /* 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;
 }


/* bucle para hacer tiempo */
 for (n=0;n<400;n++)
 {

 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(float)*dato.lon*LON_COL,tag,0,0);
 mfc_write_tag_mask(1<<tag);
 mfc_read_tag_status_all();


/* Trae los parametros de la mem principal (dato.op2) al SPE (op2)*/

 mfc_get(op2,dato.op2,sizeof(float)*LON_FIL*LON_COL,tag,0,0);
 mfc_write_tag_mask(1<<tag);
 mfc_read_tag_status_all();

```

```
     /* Operaciones a realizar */
        for (i=0;i<LON_FIL;i++)
    {
          for (j=0;j<LON_COL;j++)
          {
            valor=0;
            for (k=0;k<LON_FIL; k++)
              valor=valor+op1[i*LON_COL+k]*op2[k*LON_COL+j];
            res[i*LON_COL+j]=valor;
          }
    }

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

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

 mfc_tag_release(tag); // libera la etiqueta de transaccion
} // fin repetiones de hacer tiempo

 return 0;
}
```

3.2 Tareas a realizar

3.2.1 Implementación de la multiplicación de matrices vectorial

Lo primero a realizar es la implementación de matrices por el método vectorial explicado en la introducción. De esta manera dejamos preparado el código para su implementación paralela. Se miden tiempos y se contesta a las siguientes preguntas:

¿Tarda lo mismo que el método tradicional?

¿Por qué?

3.2.2 Aumento del rendimiento con el uso de varios SPE

Se paraleliza el código anterior y se mide 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?

3.2.3 Aumento de rendimiento con el uso de instrucciones SIMD

En este último apartado deberemos sustituir las operaciones vectorizables por la instrucción vectorial correspondiente. (Lo único que cambia es el programa del SPE).

La instrucción de multiplicación vectorial es *spu_mul(a,b)* donde *a* y *b* son vectores. Como necesitamos multiplicar escalar por vector debemos convertir el escalar en vector mediante la operación *spu_splats(s)* donde *s* es un escalar y el resultado un vector del mismo tipo que el escalar. Dado que la multiplicación vectorial de

matrices utiliza una multiplicación seguida de una suma, conviene utilizar una instrucción SIMD que ya realiza estas dos cosas de forma optimizada, se trata de *spu_madd(a,b,c)* que multiplica los vectores *a* y *b* y suma el resultado al vector *c*. No hay que olvidar incluir la cabecera *<spu_intrinsics.h>* al inicio del programa. La otra modificación a realizar consiste en definir las 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?

¿Por qué la implementación con instrucciones SIMD escala peor que la escalar?

4. Bibliografía

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

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)**