Programación de sistemas basados en el Cell

José Miguel-Alonso
Carlos Pérez Miguel

Informe Interno
EHU-KAT-IK-03-09
Departamento de Arquitectura y
Tecnología de Computadores
UPV/EHU

Resumen

La reciente aparición de procesadores de consumo con multiples cores ha puesto de manifiesto la necesidad poner en práctica tecnicas de programación paralela en el desarrollo de aplicaciones cientificas con el objetivo de aprovechar estas arquitecturas. Entre estas nuevas arquitecturas una sobresale de las demás. Este nuevo procesador creado por IBM, Toshiba y Sony lleva por nombre Cell Broadband Engine y cuenta con multiples procesadores vectoriales cuyo uso puede acelerar considerablemente muchos algoritmos. Sin embargo el uso de este procesador por su peculiar arquitectura y modelo de memoria hace de esta tarea un arduo trabajo. En este informe intentamos aproximar al programador al procesador Cell y dar algunos consejos útiles durante el desarrollo con el mismo.

1. Ejecución de programas sencillos

El SDK de Sony para la PS3 está basado en las herramientas de GNU. Más concretamente, en la familia de compiladores GCC. Vamos a limitarnos a la programación en C/C++.

Las herramientas concretas que vamos a usar son estas:

- ppu-gcc
- spu-gcc
- embedspu.sh

Recordemos que el Cell es un sistema heterogéneo, con procesadores con diferentes juegos de instrucciones. Por esta razón tenemos que usar distintos compiladores.

Supongamos que tenemos un programa en C "convencional", no diseñado específicamente para el Cell. Podemos compilarlo y ejecutarlo de distintas formas. Partamos del clásico "hello_world.c":

```
#include <stdio.h>

main (int argc, char *argv[]) {
  printf("Hello world!\n");
  return(0);
}
```

Si lo compilamos con gcc, usando la orden habitual gcc −o hello hello_world.c obtendremos un programa "hello" ejecutable en la PPU. Alternativamente, podríamos haber utilizado ppu−gcc −o hello_ppu hello_world.c. El resultado sería prácticamente el mismo: en ambos casos utilizaríamos un compilador gcc, versión 4.1.1. Las diferencias están en las configuraciones del compilador, al construirlo.

Si queremos que nuestro programa se ejecute en una SPU, la orden de compilación sería spu−gcc −o hello_spu hello_world.c. El resultado: un programa ejecutable en la PPU que prepara el entorno para que una SPU ejecute nuestro programa, y que le da auxilio para que las llamadas al sistema se redirijan adecuadamente. A efectos prácticos, nuestro programa principal se ejecuta, como queríamos en una SPU.

En general, estas aproximaciones no son prácticas. Lo habitual es que el programador sea consciente de que trabaja en un entorno mixto y prepare programas fuenteseparados. Veamos una versión "Cell" de nuestro saludo al mundo. Empezamos por el programa "spu_hello.c", destinado a ejecutarse en una o más SPUs:

```
#include <stdio.h>
int main(unsigned long long spuid){
  printf("Hello, World! (From SPU: %llx)\n",spuid);
  return (0);
}
```

Es muy similar al "hello_world.c" de antes, salvo que en la orden de impresión hemos incluido el identificador de SPU.

Por otra parte, preparamos el programa "ppu_hello.c", para su ejecución en la PPU. Su trabajo ahora es de organización: prepararlo todo para que las seis SPUs disponibles en la PS3 ejecuten el programa anterior:

```
#include <stdio.h>
#include <libspe.h>
#define SPU_NUM 6
extern spe_program_handle_t spu_hello_handle;
int main(void) {
  int status,i;
  int* speid[SPU_NUM];
  for(i=0;i<SPU_NUM;i++)
    speid[i] = spe_create_thread (0, &spu_hello_handle, NULL, NULL, -1, 0);
  for(i=0;i<SPU_NUM;i++)
    spe_wait(speid[i], &status, 1);
  return 0;
}
```

Como se puede ver, el trabajo de este código es preparar 6 hilos de ejecución (uno por SPE), indicar que el programa a ejecutar se obtiene a través del

handle “spu_hello_handle”. Ese handle va a ser el nexo de unión entre los dos programas.

Para compilar y ejecutar esta aplicación mixta, es necesario dar varios pasos. Empezamos compilando el código para las SPUs

```
spu−gcc −o spu_hello spu_hello.c
```

En este caso (aunque no en todos) podemos comprobar que el programa funciona, ejecutándolo directamente. El resultado sería algo como esto:

```
Hello, World! (From SPU:100000000)
```

Esto no es, sin embargo, lo que queríamos. El segundo paso es obtener, a partir del ejecutable anterior, un programa objeto que incluya el código para las SPUs, y que pueda ser en tiempo de ejecución usado por la PPU para enviarlo a las SPUs y ejecutarlo allí. En definitiva, vamos a preparar un fichero que contendrá el código a ejecutar por los threads lanzados desde la PPU (que se ejecutarán realmente en las SPUs). La orden es la siguiente

```
embedspu.sh −m64 spu_hello_handle spu_hello spu_hello_ld.o
```

Fijémonos en los tres últimos argumentos. “spu_hello” es el programa ejecutable en las SPUs. A partir de él, obtenemos “spu_hello_ld.o”, que ya no es ejecutable directamente, pero que puede ser utilizado desde la PPU, a través del handle “spu_hello_handle”.

Ahora queda montar la aplicación. El programa principal será el código para la PPU, que tendrá acceso al código a ejecutar en las SPUs. La orden es esta:

```
ppu−gcc −lspe −o cell_hello spu_hello_ld.o ppu_hello.c
```

Nótese cómo se utiliza la librería “spe”, que incluye las funciones spe_thread_create() y spe_wait(). También se incluye el módulo objeto spu_hello_ld.o que contiene el programa a ejecutar en las SPUs, al que accede a través del handle preparado con el programa embedspu y conocido por la PPU.

Por fin tenemos el programa ejecutable. Lo comprobamos:

```
[PS3]$ ./cell_hello
Hello, World! (From SPU:10018060)
Hello, World! (From SPU:10018b50)
Hello, World! (From SPU:10018e30)
Hello, World! (From SPU:10019110)
Hello, World! (From SPU:100193f0)
Hello, World! (From SPU:100196d0)
```

Resumiendo, la compilación se ha hecho así:

```
spu−gcc −o spu_hello spu_hello.c
embedspu.sh −m64 spu_hello_handle spu_hello spu_hello_ld.o
ppu−gcc −lspe −o cell_hello spu_hello_ld.o ppu_hello.c
```

2. libspe vs libspe2

En el programa anterior hemos incluido <libspe.h> y compilado con −lspe. En ambos casos nos estamos refiriendo a la librería de integración PPU-SPU

denominada SPE 1.2. Existe otra variante, más moderna, denominada SPE 2.0. Las diferencias entre ambas son importantes, a nivel de uso. Mientras SPE 1.2 está pensada para entornos Unix, y asocia un hilo de ejecución de una SPU a un thread Posix que se ejecuta en la PPU, SPE 2.0 ofrece un entorno de gestión de trabajos en las SPUs independientes del S.O. de base.

En SPE 2.0

El programa anterior, para su ejecución con SPE 2.0, necesitaría algunas modificaciones en la parte PPU, y diferentes órdenes de compilación. El código para la PPU quedaría como sigue (ppu_hello_v2.c):

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <libspe2.h>

#define NUM_SPUS 6

extern spe_program_handle_t spu_hello_handle;
void *launcher(void *data);

int main() {
   pthread_t my_thread[NUM_SPUS];
   int retval, i;

   for (i=0; i<NUM_SPUS; i++) {
      retval = pthread_create(&(my_thread[i]), NULL, launcher, NULL);
      if(retval) {
         fprintf(stderr, "Error creating thread! Exit code is: %d\n", retval);
         exit(1);
      }
   }
   for (i=0; i<NUM_SPUS; i++) {
      retval = pthread_join(my_thread[i], NULL);
      if(retval) {
         fprintf(stderr, "Error joining thread! Exit code is: %d\n", retval);
         exit(1);
      }
   }
   return 0;
}

void *launcher(void *data) {
   int retval;
   unsigned int entry_point = SPE_DEFAULT_ENTRY;
   unsigned int flags = 0;
   spe_context_ptr_t my_context;

   my_context = spe_context_create(SPE_EVENTS_ENABLE|SPE_MAP_PS, NULL); // Creacion cont

   spe_program_load(my_context, &spu_hello_handle); // Carga de programa en el LS de una SPU
   do {
      retval = spe_context_run(my_context, &entry_point, 0, NULL, NULL, NULL); // Ejecucion
   } while (retval > 0);
   spe_context_destroy(my_context); //Destruccion contexto
```

```
    pthread_exit(NULL);
}
```

Lo más relevante: es el programador el que gestiona los hilos en la PPU. Es imprescindible que haya hilos, porque la función spe_context_run() es síncrona: nos quedamos bloqueados en ella. En cuanto a la compilación, sólo cambia el montaje final:

```
ppu-gcc -lspe2 -o cell_hello_v2 spu_hello_ld.o ppu_hello_v2.c
```

3. Comunicación y sincronización

3.1. Modelo de memoria

De forma muy simplificada: existe un MS (espacio de almacenamiento principal), accesible sin restricciones por el PPE. Sin embargo, los SPEs no pueden acceder directamente al MS. Cada SPE tiene su propio LS (espacio de almacenamiento local), de tan sólo 256 KB, en el que se almacena código y datos.

Por lo tanto, los SPEs no tienen un espacio de memoria compartido, y es necesario realizar copias de información (vía DMA) entre MS y los LSs. Para comunicación a pequeña escala, existen mecanismos como las colas de mensajes (mailboxes) y las señales.

Cada SPE contiene una unidad MFC (Memory Flow Controller) encargada entre otras cosas de gestionar las transferencias DMA al LS. Cada SPU se pone en contacto con su MFC usando canales (channels).

3.2. Comunicación entre procesadores

Hay tres mecanismos básicos de comunicación entre procesadores:

Mailboxes: colas para el intercambio de mensajes de 32 bits. Cada SPU tiene un "SPU Write Outbound Mailbox" y un "SPU Write Outbound Interrupt Mailbox" para enviar mensajes al PPE, además de un "SPU Read Inbound Mailbox" para recibir mensajes. Más adelante vemos un ejemplo de uso de mailboxes para sincronización PPU-SPUs.

Signal notification registers: cada SPU tiene dos registros de 32 bits para recibir notificaciones. Cada uno tiene un registro MMIO asociado en el cual el procesador emisor puede escribir los datos de la notificación. Tanto el PPE como los SPEs pueden enviar señales.

DMA: el mecanismo principal para transferir datos entre el MS (espacio de almacenamiento principal) y el LS (espacio de almacenamiento local) asociado a cada SPU.

3.3. Transferencia DMA entre MS y LS

La transferencia entre el MS y el LS de un SPE particular puede ser iniciada desde el SPE, o desde el PPE. En cualquier caso, el control lo va a llevar el MFC del SPE afectado.

Cada MFC puede gestionar varias transferencias DMA en curso, así como recibir listas de transferencias que irá procesando adecuadamente. Cada comando para DMA está etiquetado con un "Tag Group ID" de 5 bits. Este identificador sirve para comprobar si una operación (o grupo) ha finalizado.

Las transferencias DMA contienen dos direcciones: LSA (dirección válida en el LS) y EA (dirección efectiva, válida globalmente, y que incluye todo el MS). El MFC realiza transferencias de 1, 2, 4 y 8 bytes, así como de múltiplos de 16 bytes. El tamaño máximo de una transferencia es 16 KB. El rendimiento máximo se obtiene cuando las direcciones local y efectiva (LSA y EA) están alineadas a 128 bytes, y el tamaño del bloque a transferir es múltiplo par de 128 bytes.

La comunicación con el MFC para ordenar operaciones de DMA (y otras) se hace mediante el envío de comandos. Hay dos formas diferentes de enviar comandos, que dependerán de quién los inicie:

- Una SPU envía comandos a su MFC local mediante instrucciones sobre canales (read channel, write channel, read channel count).
- La PPU u otra SPU puede enviar comandos a un MFC no local mediante operaciones de acceso (load, store) en I/O mapeado en memoria (MMIO)

Un MFC mantiene dos colas de comandos separadas, una para comandos recibidos de su SPU (mediante canales) y otra para comandos recibidos de fuera (mediante MMIO).

3.4. Ejemplo de sincronización con mailboxes

Veamos un programa mixto PPU-SPU que utiliza las SPUs para generar números aleatorios (usando una librería libmisc). Los mailboxes se usan para que cada SPU avise a la PPU cuando ha terminado su trabajo.

Código PPU

```
/* author: Guochun Shi gshi@ncsa.uiuc.edu */

#include <stdio.h>
#include <errno.h>
#include <libspe.h>
#include <sys/time.h>
#include "common.h"

extern spe_program_handle_t simple_spu;

#define SPU_THREADS 6

int main(int argc, char **argv) {
    speid_t spe_ids[SPU_THREADS];
    int i;
    int g_spe_threads = SPU_THREADS;
    struct timeval t0, t1;

    for(i=0; i<g_spe_threads; i++) {
        spe_ids[i] = spe_create_thread(0, &simple_spu, 0 , NULL, -1, SPE_MAP_PS);
        if (spe_ids[i] == 0) {
```

```
            fprintf(stderr, "Failed spu_create_thread(rc=%p, errno=%d)\n", spe_ids[i], errno);
            exit(1);
        }

    }

    gettimeofday(&t0, NULL);
    for (i = 0; i < g_spe_threads; i++) {
        spe_write_in_mbox(spe_ids[i],1);
    }

    for (i =0; i < g_spe_threads; i++) {
        while(spe_read_out_mbox(spe_ids[i])!= 1) {;}
    }

    gettimeofday(&t1, NULL);
    double timecost = (t1.tv_sec - t0.tv_sec) + 0.000001*(t1.tv_usec -t0.tv_usec);
    printf(" rate = %f millilon/sec\n", 1.0*COUNT*8/timecost/1000000);
    return (0);
}
```

El fichero de cabecera "common.h" contiene únicamente esta declaración

```
#define COUNT (8*1024*1024)
```

El código genera, usando libspe, tantos hilos como SPUs. El programa a ejecutar por esos hilos se obtiene a través del handle "simple_spu". Tras ello, se obtiene el contenido del reloj del sistema, y se escribe en los buzones de los hilos una especie de señal de "go!". Hecho esto, se espera de todos y cada uno de los hilos, también vía buzón, una señal de "done!" y se vuelve a obtener la hora. El programa termina con un cálculo de la cantidad de números aleatorios generado en el intervalo de tiempo entre medidas.

Nótese el uso de las funciones de escritura en los buzones "in" de las SPUs (spe_write_in_mbox()) para enviar señales, y de lectura de los buzones "out" de las SPUs (spe_read_out_mbox()) para recibir señales. Estas funciones también corresponden a libspe.

Código SPUs

El programa que ejecutan las SPUs es el siguiente ("simple_spu.c")

```
/* Guochun Shi gshi@ncsa.uiuc.edu */
#include <libmisc.h>
#include <spu_mfcio.h>
#include "common.h"

vector float f1, f2;

static void generate_hand_v(void) {
    f1 = rand_0_to_1_v();
    f2 = rand_0_to_1_v();
    return;
}

int do_compute_db() {
```

```
    int i;
    for (i=0; i<COUNT; i++) generate_hand_v();
    return 0;
}

int main ( unsigned long long spu_id, unsigned long long parm) {
    spu_read_in_mbox();
    do_compute_db();
    spu_write_out_mbox(1);
    return 0;
}
```

Lo de menos en este ejemplo es que cada SPU genera "COUNT" veces dos vectores de números aleatorios en coma flotante (en el rango $[0, 1)$). Nos interesa ver cómo se espera a la señal de "go!" con la función spu_read_in_mbox(), y cómo se envía la señal de "done!" escribiendo un "1" en el buzón "out" con la función spu_write_out_mbox(). Estas funciones están definidas en "spu_mfcio.h" Compilación y ejecución Compilamos el programa de las SPUs. Necesitamos acceder a la librería libmisc del SDK de IBM. Para ello usamos el flag −L.

```
spu−gcc −o simple_spu simple_spu.c −L/usr/spu/lib/sdk −I/usr/spu/include −lmisc
```

Preparamos el fichero .o montable con la aplicación.

```
embedspu.sh −m64 simple_spu simple_spu simple_spu_ld.o
```

Y montamos la aplicación completa. El ejecutable mixto queda en main.

```
ppu−gcc −lspe −o main main.c simple_spu_ld.o
```