Escola Politècnica Superior de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Cluster d'ordinadors per estacions base

TITULACIÓ: Enginyeria Tècnica de Telecomunicació, especialitat Sistemes de Telecomunicació

AUTOR: Albert Paños Marín

DIRECTOR: Antoni Gelonch

DATA: 3 de Febrer de 2011

Títol: Maqueta de TFC/PFC

Autor: Albert Paños Marín

Director: Antoni Gelonch

Data: 3 de Febrer de 2011

Resum

Es planteja el desenvolupament d'un cluster d'ordinadors que pugui dur a terme la computació intensiva necessària en una estació base.
El computador de base a analitzar es la PS3 utilitzant les eines de programació disponible i sota entorn Linux.

Les activitats que es duran a terme son:
-Iniciació a l'entorn de desenvolupament per la PS3.
-Anàlisi de les prestacions que es poden aconseguir i identificació de les limitacions en el desenvolupament de codi paral-lelitzable.
-Definició i posada en marxa d'un cluster de almenys 2 PS3.
-Introducció de l'etapa de gestió de bits de l'estàndard WiMAX sota execució PS3.
-Anàlisi de prestacions.

Title: TFC/PFC Model

Author: Albert Paños Marín

Director: Antoni Gelonch

Date: 3 February of 2011

Overview

It describes the development of a cluster of computers that can perform the Intensive computation needed in a base station.
The computer analysis is based on the PS3 using the programming tools Available under Linux environment.

The activities carried out are:
-Initiation of development environment for the PS3.
-Analysis of the benefits that can be achieved and identify constraints in the development of parallel code.
-Definition and implementation of a cluster at least of 2 PS3.
-Entering the stage management of the standard WiMAX bit execution under PS3.
-Analysis of benefits.

ÍNDEX

INTRODUCCIÓ **1**

OBJECTIUS **2**

ACRONIMS **3**

CAPÍTOL 1. PLAYSTATION 3 ..... **4**

1.1. Informació tècnica **4**
1.1.1. IBM Cell/BE 5
1.1.2. Power Processor Element 6
1.1.3. Synergistic Processor Element 7
1.1.4. Element Interconnect Bus 9
1.1.5. Main Memory System 10

1.2. Comparativa processadors actuals **11**

CAPÍTOL 2. PROGRAMACIÓ EN PS3 ..... **12**

2.1. Prerequisits **12**

2.2. Tipus de programació **12**
2.2.1. Programació lineal o seqüencial 12
2.2.2. Programació concurrent 12
2.2.3. Programació paral·lela 13

2.3. SIMD **13**

2.4. Transferència de dades **14**
2.4.1. Transferència iniciada per SPE 15
2.4.2. Transferència iniciada pel PPE 16
2.4.3. Alineament i mida de les dades 16
2.4.4. Double buffering: 16
2.4.5. Mailboxes 18

2.5. Programes SPE Embedded **20**

CAPÍTOL 3. MULTIPROCESSAMENT SOBRE PS3 ..... **21**

3.1. Introducció **21**

3.2. Llibreries **21**

3.3. Estructura programa multiprocessament **21**

3.4. Declaració variables a memòria **22**

3.5. Funcions per multiprocessament en CBEA **23**

3.6. Transmissió entre SPE **24**

3.7. Funcions SIMD per realitzar operacions vectorials **26**

CAPÍTOL 4. RESUM 27

CAPÍTOL 5. RESULTATS I CONCLUSIONS 31

5.1. Resultats 31

5.2. Conclusions 33

ANNEXES. 35

Instal·lació software a PS3 35

Instal·lació software i SDK a CPU 35

Codi generat per multiprocessament 36

BIBLIOGRAFIA 49

INTRODUCCIÓ

En l'actualitat, s'intenten trobar noves innovacions tecnològiques als camps que més utilitat tenen a la nostra vida. Entre aquestes àrees podem trobar totes les tecnologies referents a comunicacions sense fils, i la creació de processadors amb múltiples nuclis.

En la comunicació sense fils, s'han realitzat varies evolucions amb el sistema Wireless LAN (WLAN). Aquestes implicaven augment del radi d'acció o taxa de bits enviats per segon. Amb l'aparició de dispositius mòbils que es poguessin connectar a Internet s'estan intentant creant nous estàndards de nova generació, com en aquest cas: WiMAX .

Referent a la millora dels processadors s'han obtingut grans avenços. Primer va evolucionar la velocitat d'un únic processador. Posteriorment, s'ha intentat reduir la càrrega dels processadors, creant ordinadors amb múltiples processadors que realitzen tasques paral·lelament. Això implicava que, encara que el processador per si sol tenia una capacitat de processament limitada, quan s'executaven diversos procediments, s'executaven en diferents processadors y així es podien realitzar varies tasques tenint un rendiment major.

El treball que ens ateny, es un híbrid entre aquest dos àmbits. Es farà servir un processador múltiple (una PlayStation 3) , per investigar l'optimització de codis que es puguin separar en mòduls independents. Realitzant aquesta tasca, aconseguiríem poder realitzar diferents branques independents d'un procés que convergeixen les seves dades en un procés posterior. Amb això obtindrem una eficiència molt més gran que la creació de processos paral·lels en un mateix processador (posat que la memòria es compartida).

La primera part es centrarà en una explicació tècnica de la PlayStation 3 y la comparació amb alguns processadors d'ordinadors actuals. Es vol demostrar que suposaria substituir un processador convencional per un multiprocessador i quins avantatges obtindríem.

La segona part serà la introducció al món de la programació amb el SDK d'IBM. Aquest software ens permet introduir certes comandes específiques per potenciar la multi tasca de la PlayStation 3 y així obtenir una resposta, de certs sistemes, molt més eficient. A més, s'exposaran les bases de l'arquitectura dels diferents tipus de programació.

La tercera part és la més rellevant de totes. En aquesta secció comentarem pas a pas les diferents parts del programa que es vol crear per realitzar el multiprocessament. A cada apartat es comentarà el cas genèric i les diferents opcions, i a més, el cas particular que a nosaltres hem generat.

OBJECTIUS

Així doncs en aquest treball s'intentaran tractar d'una manera més o menys tècnica les diferents opcions potencials dels processadors de la PlayStation3.

1. Coneixement molt més ampli de l'estructura del processador de la PlayStation 3.

2. Assoliment del coneixements de programació que fan possibles una major eficiència del sistema

3. Correcta instal·lació del software necessari per realitzar les tasques de programació

4. Realització de programa base que realitzi transacció d'informació entre diferents processos.

5. Ajust dels diversos paràmetres de les funcions per aconseguir un programa segons uns criteris.

Posteriorment, es voldria poder adaptar un codi que simula una arquitectura de transmissió WiMAX per tal de realitzar una execució molt més eficient d'aquest procés.

ACRONIMS

PPE: Power Processing Element
SPE: Synergistic Processing Element
SPU: Synergistic Processing Unit
LS: Local Store
MFC: Memory Flow Controller
EIB: Element Interconnection Bus
MIC: Memory Interface Controller
MMS: Main Memory System
SP: Single Precision
DP: Double Precision
CBEA Cell Broadband Engine Arquitecture
LSA Local Start Address
EA Effective Address
NaN Not a Number
SIMD Single Instruction, Multiple Data
WiMAX Worldwide Interoperability for Microwave Access
PS3 PlayStation 3
SDK Software Development Kit

CAPÍTOL 1. PlayStation 3

La PlayStation 3 es la tercera videoconsola de sobretaula de Sony Computer Entertainment i es part de la setena generació de videoconsoles.
Va sortir al mercat 11 de novembre de 2006 al Japó i posteriorment el 23 de març de 2007 a Europa. En aquest moment es podien adquirir dues versions de diferents capacitats de disc dur, de USB i retrocompatibilitat amb la PlayStation 2.

1.1. Informació tècnica

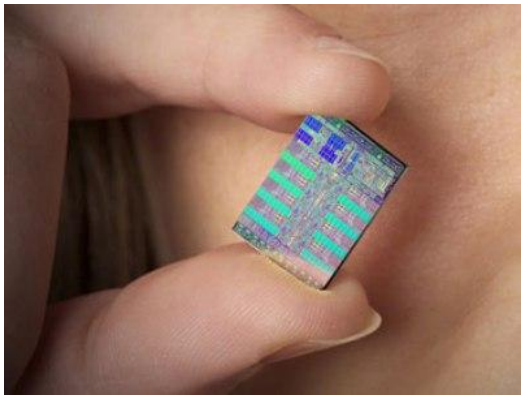
Fig.1.1

- CPU: IBM Cell/BE a 3.2 GHz
  - 218 GFLOPS
  - RAM principal: 256 MB XDR (64b cada SPE a 3.2 GHz) a 25,6 GB/s
- GPU: Nvidia RSX
  - 1.8 TFLOPS
  - VRAM: 256 MB GDDR3 (2 x 128b a 700 MHz)
- Bus del sistema a 2.5 GB/s

- Dos ports USB 2.0 frontals
- Connexió Ethernet: 10Base-T, 100Base-Tx, 1000Base-T
- Connexió Wi-Fi 802.11 de la classe b/g
- Connexió Bluetooth 2.0
- Video out: HDMI, Component, Composite, RGB, Scart.

1.1.1. IBM Cell/BE

El nucli bàsic de computació de la PS3, es un processador d'IBM del tipus Broadband Engine. Aquest processador esta compost per:



- PPE
- EIB
- SPE
  - SPU
  - LS
  - MFC
- BEI
- MMS

Fig.1.2

L'arquitectura de blocs d'aquest processador es podria representar pel diagrama següent.

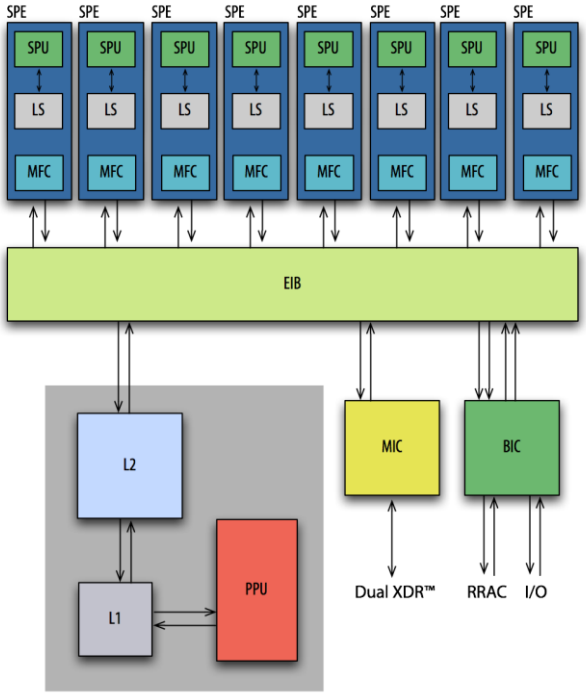


Fig.1.3

1.1.2. Power Processor Element

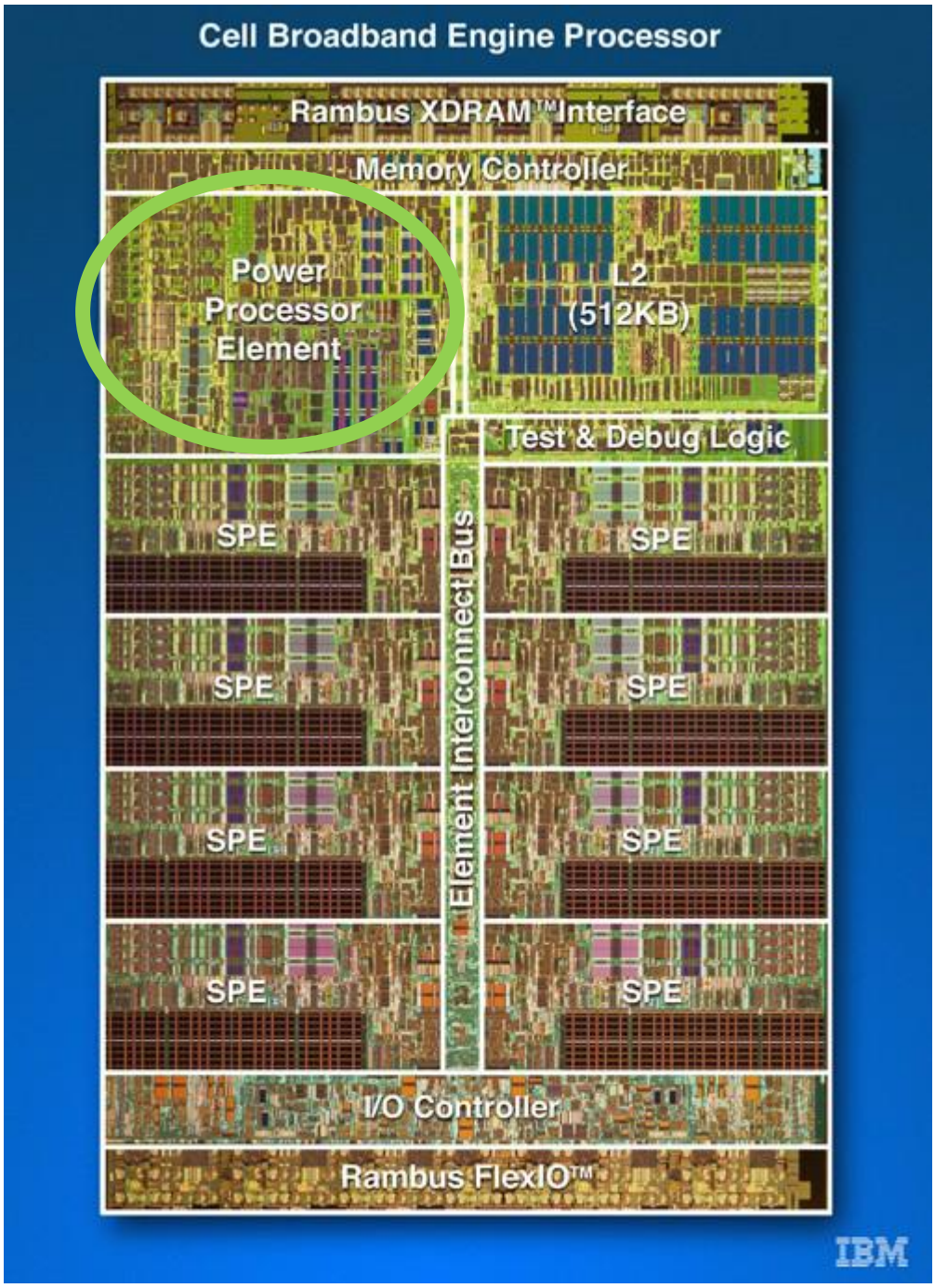


Fig.1.4

Aquest mòdul es la base del processador CBE. El PPE es un processador de 64 bits amb una Arquitectura Power PC Dual-thread. Consta d'una memòria 32KB L1 cache i una altra de 512 KB L2.

Aquest processador té unes instruccions específiques de programació com són les de operacions amb vectors i les SIMD multimèdia ("Altivec") per una banda, i les instruccions de comunicació entre el PPE i els SPE's al seu voltant.

1.1.3. Synergistic Processor Element

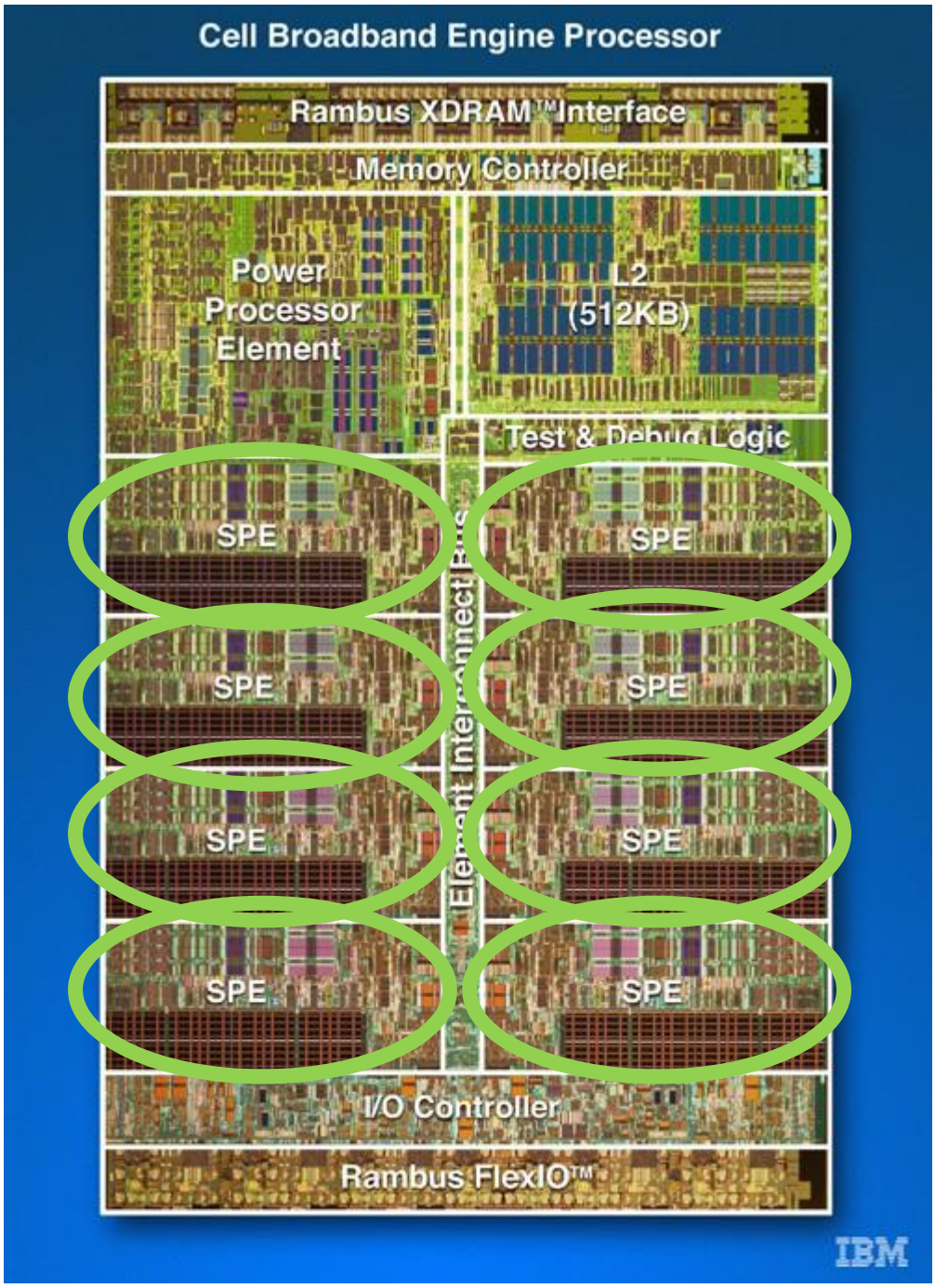


Fig.1.5

En aquest processador CBE existeixen 8 SPE, encara que només es fan servir 6 (ja que un s'inhabilita per qüestions de fabricació i l'altre el fa servir hypervisor). Aquest SPE són uns processadors auxiliars que realitzen les tasques que li encomana el PPE. Així doncs, el PPE gestiona la creació de subtasques i un dels SPE la realitza.

Aquest SPE esta format per un nucli RISC, amb una memòria RAM local de 256Kb y 128 bits de fitxers de registres. Com passa amb el cor del CBE, aquest

també consta d'unes instruccions específiques: operacions amb vectors, control DMA i missatges i sincronia amb els altres processadors.

La freqüència de treball en sistema SP es de 3,2 GHz, a una velocitat de 25,6 Gflops/s (totalment pipeline), Si es troba en sistema DP a la mateixa freqüència de rellotge, adquireix una velocitat de 1,8 Gflops/s (amb 7 cicles de latència).

Els registres al SPE són del 0 al 127 de propòsit general (GPR), i el 128 és de estatus de coma flotant i control de registre (FPSCR).

Com tot processador, consta d'uns petits retards per d'utilització de diferents instruccions. Així doncs:

Taula.1.1

Operació	Cicle de retard
Coma fixa simple	2
Coma fixa complexa	4
Inici	6
Coma flotant simple	6
Multiplicació d'enters	7
Branch miss	20
DP flotant (parcialment pipeline)	13
Comanda DMA de pregunta	20

Sobretot, a diferencia del PPE, el SPE no pot albergar un sistema operatiu al seu interior. Aquest només realitza les tasques que li encomana el PPE, mai pot decidir quines instruccions realitzar.

1.1.3.1. Single Precision

En aquest mode de funcionament, a cada cicle l'SPE pot realitzar quatre elements d'un vector i dos operacions en cada un d'aquests elements. Aquests processos són els mateixos que realitza el PPE.
Així doncs, es poden obtenir en SP les següents velocitats:
8x4x2x3.2GHz = 204.8 Gflops/s
4x2x3.2GHz = 25.6 Gflops/s
Per tant, treballa a un total de 230.4 Gflops/s.

1.1.3.2. Doble Precision

En aquest mode de funcionament cada set cicles l'SPE pot processar dos elements d'un vector y dos operacions a cada un d'aquests elements.
En canvi, el PPE treballa igual que en el mode SP. És a dir, realitza el processament de quatre elements i de dos operacions a cada un d'aquests, en un cicle.
Així doncs en aquest mode es poden obtenir les següents velocitats:
8x2x2x3.2GHz/7 = 14.63 Gflops/s
4X2x3.2GHz = 25.6 Gflops/s
Per tant, en DP, treballa a un total de 40.23 Gflops/s.

1.1.4. Element Interconnect Bus

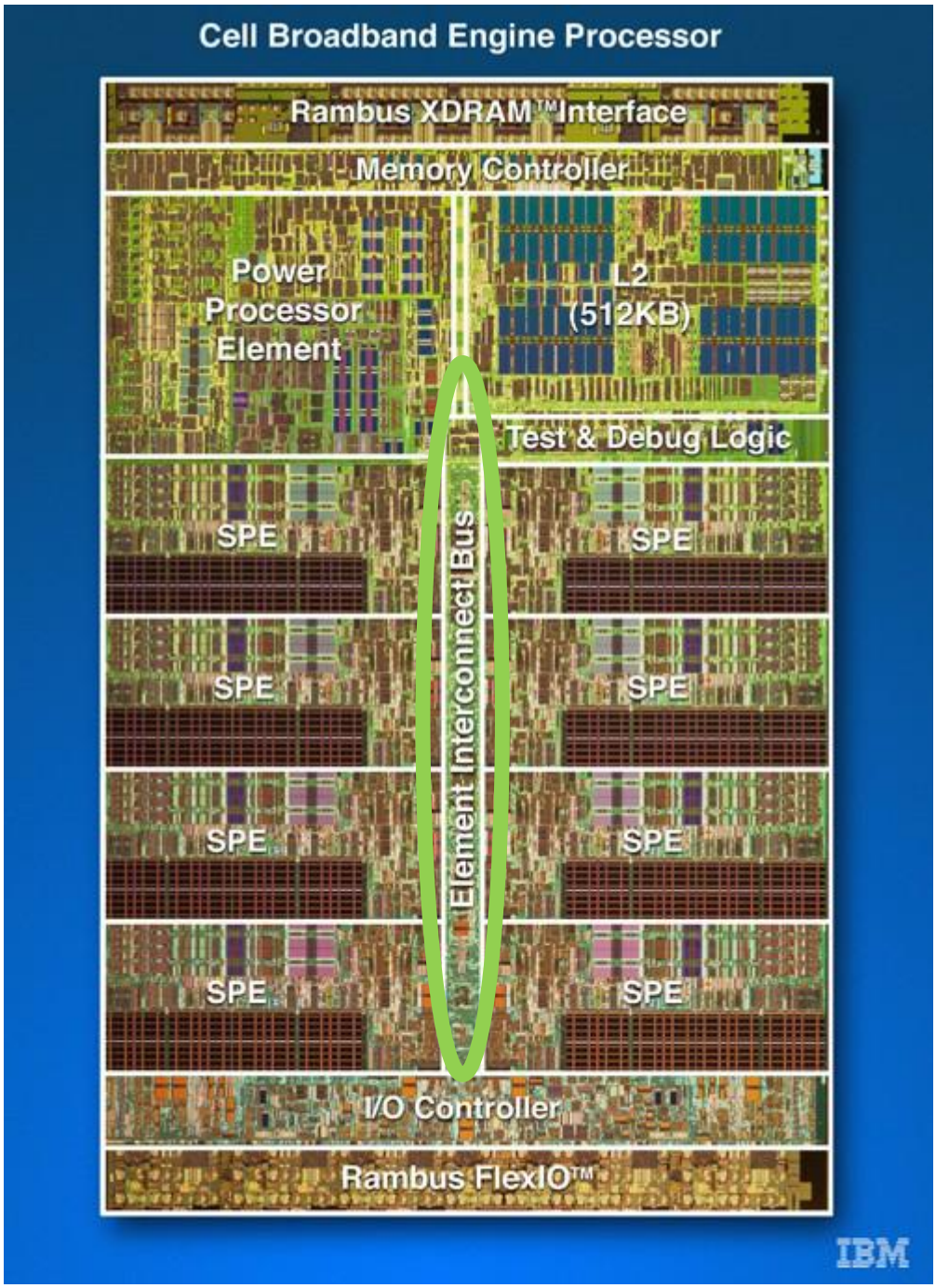


Fig.1.6

El EIB es l'element per el qual s'envia l' informació tant entre el PPE i els SPE's, com entre els SPE's.

Aquest bus, té un pic d'ample de banda de 96 bytes/cicle amb quatre anells unidireccionals de 16 bytes d'informació, que es transmeten en un ample de

banda de 204,8 GB/s, amb un rellotge de 1,6 GHz. A més, pot albergar fins a 100 respostes DMA.

1.1.5. Main Memory System

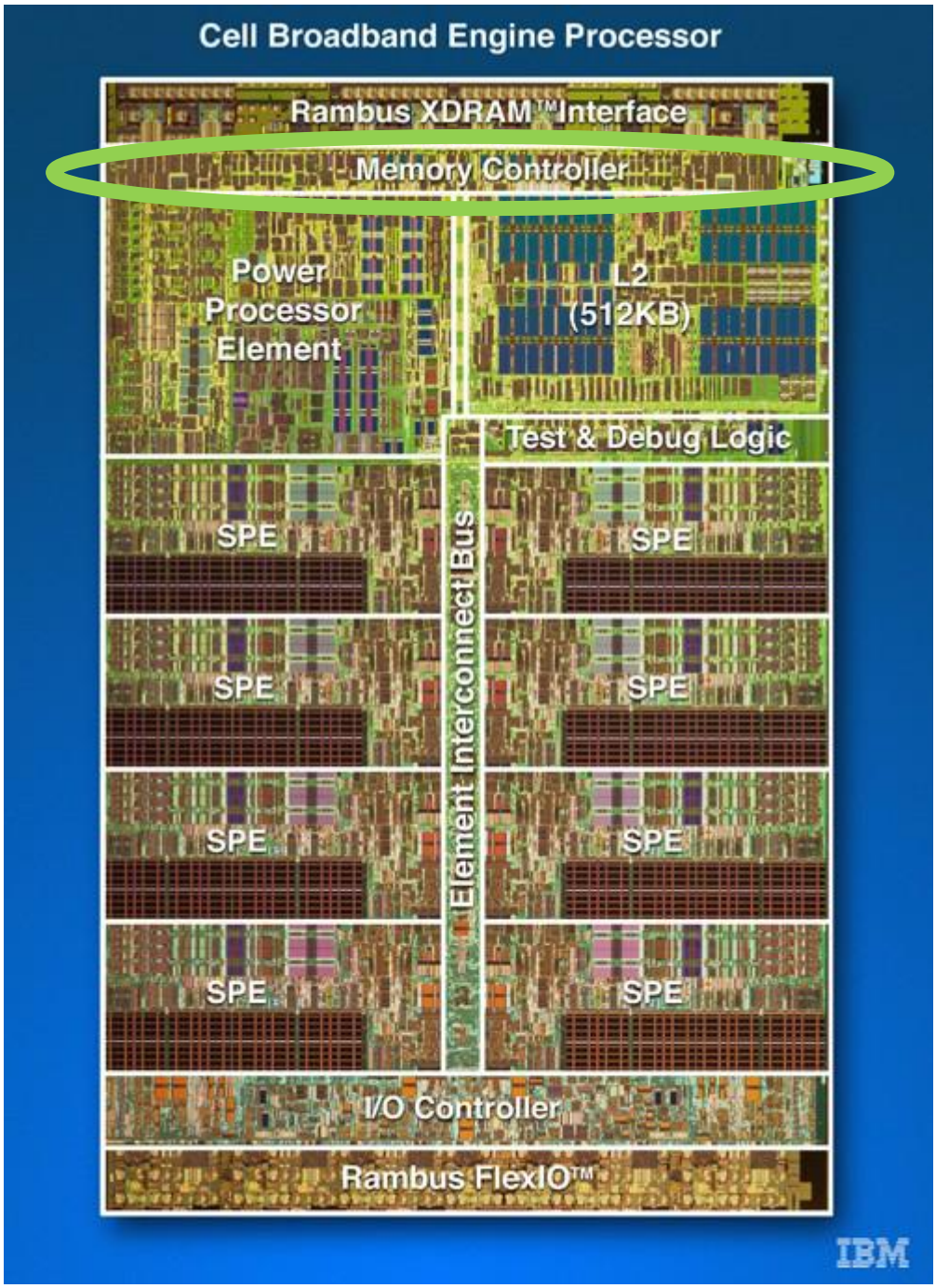


Fig.1.7

Esta format pel MIC amb sortida externa dual XDR. Aquesta memòria té una freqüència efectiva màxima de 3,2 GHz. Cada una amb 8 banks de 256 MB màxim y un total de 16 banks amb un màxim de 512 MB. El MIC treballa a una velocitat de 25,6 GB/s.

1.2. Comparativa processadors actuals

Realitzarem la mateixa comparativa que durant tot el procés: DP i SP.

En DP comparem un nucli 1.6 GHz Dual-Core Itanium 2 amb un 3.2 GHz CELL BE (només els SPE):

Taula.1.2

Itanium 2	CBE
1.6x4x2 = 12.8 Gflops/s	3.2x8x2x2/7 = 14.63 Gflops/s

En SP compararem els mateixos processadors:

Taula.1.3

Itanium 2	SPE	CBE
1.6x4x2x2 = 25.6 Gflops/s	3.2x4x2 = 25.6 Gflops/s	3.2x8x4x2 = 204.8 Gflops/s

En aquest cas podem observar com un processador CELL BE a 3.2 GHz, on només es fan servir els SPE, és el mateix que fer servir 8 processadors Dual-Core Itaniums 2 a 1.6 GHz.

Connexió màquines CM-5, que consta d'unes 512 CPU's implicaria:
512x128 = 65 Gflops/s DP
En canvi, si connectem 4 PlayStation3 obtindríem:
4x15 = 60 Gflops/s DP.
Així doncs, si comparem seria :
60 Gflops/s / 65 Gflops/s = 0.923 en conjunt, y 0.923 *(512 CPU / 4PS3) = 118.144 vegades més potent una PS3 que una CPU de128 MB.

CAPÍTOL 2. PROGRAMACIÓ EN PS3

2.1. Prerequisits

Encara que la PS3 porta el seu propi sistema de gestió de dades (hipervisor), SONY permet introduir un sistema operatiu totalment diferent. Aquest sistema operatiu de lliure distribució basat en UNIX. Entre totes les distribucions que existeixen, s'ha d'instal·lar obligatòriament Fedora. Per poder arrencar aquest sistema s'han d'instal·lar uns programes que autoarrenquen Linux des de l'hipervisor de la vídeo consola.

El següent requisit després d'instal·lar Fedora és la instal·lació del SDK de programació. Aquest paquet proporciona les eines necessàries per poder incorporar les peculiaritats de la programació orientada a la PS3, El SDK es pot fer servir en els llenguatges de programació C, C++ i Fortran. Una mica més endavant s'explicaran aquestes peculiaritats.

Una vegada fet això podem començar a utilitzar la PS3 de processador, ja que SONY no ha permès accedir a la part gràfica d'aquesta.

2.2. Tipus de programació

2.2.1. Programació lineal o seqüencial

Aquest tipus de programació es la clàssica. En aquesta tenim un fil conductor i no es realitzen salts d'una part del codi a una altra.

Ex:

```
j informada en un scanf;
i = 2;

if ( i + j > 5 )
{
// Codi
};
```

2.2.2. Programació concurrent

És la següent evolució en la programació. En aquest tipus és una execució simultània de codis seqüencial en diferents parts d'un mateix processador o directament en processadors diferents.

Aquest tipus de codi és exactament igual al seqüencial, però s'executa una comanda específica que realitza un fill per poder processar un codi.

2.2.3. Programació paral·lela

Aquest tipus de programació és el que actualment esta vigent. Es realitza el processat d'un mateix codi, o diferents codis, però en processadors totalment diferents.

Per exemple en el cas que ens toca a nosaltres, tenim un PPE que executa un codi principal. Aquest codi principal crea fills que es processen en els SPE, però de manera simultània.

Amb això el que obtenim és un rendiment molt més gran, però necessitem un control del flux de dades més controlat. Això es degut a que si es realitzen càlculs de manera simultània, però no sincronitzada, potser en quan s'acabi un procés encara no tenim les dades necessàries pel següent, perdent eficiència.

2.3. SIMD

Les funciones matemàtiques tradicionals especificades pels estàndards com ISO/IEC 9899:1999 (estàndard C99) estan definits en termes d'instruccions escalars i no agafen els avantatges de les funciones SIMD. Instruccions vectorials proporcionades pel paquet d'instruccions pel PPU i el SPU de l'arquitectura Cell BE.

La llibreria matemàtica SIMD proveeix versions curtes de vectors d'un subpaquet de les funciones matemàtiques tradicionals. Aquestes versions dels vectors normalment son el més tancades possibles segons les especificacions fixades pels estàndards escalars. Però, les diferencies fundamentals entre l'arquitectura escalar i l'arquitectura Cell BE requereixen algunes desviacions, incloent el tractament de l'arrodoniment, condicions d'error, excepcions de coma flotant i operacions especial com NaN i infinitius.

Aquesta llibreria esta proveïda en el SDK tan com llibreria linkable com funcions de capçalera. Per diferenciar els noms de funcions amb els seus escalars per definir un vector, se'ls posa als primers una barra baixa al començament. Un altre peculiaritat d'aquestes llibreries es que, a més de tenir que fer servir la llibreria simdmath.h, per cada funció no lineal que es faci servir cal introduir la seva llibreria específica (Ex: fabsf4.h).

La avantatja de fer servir aquest tipus de comandes, és bàsicament de rendiment. Això es degut principalment, a fet de ser dades vectorials. Amb aquest tipus de comandes es pot realitzar el càlcul de certs tipus d'operacions molt més ràpid. Aquest tipus d'operacions poden ser tant aritmètiques com lògiques o barrejades.

L'inconvenient de l'ús d'aquestes comandes es la necessitat d'homogeneïtzar les operacions. És a dir, només es poden fer de cop el mateix tipus d'operació. Aquesta tasca no s'encarrega de fer-la ni el processador ni el linker. Per tant el programador ha de tenir un cert ordre al realitzar el codi, ja que així podrà fer servir aquestes comandes sense tenir que remodelar el codi.

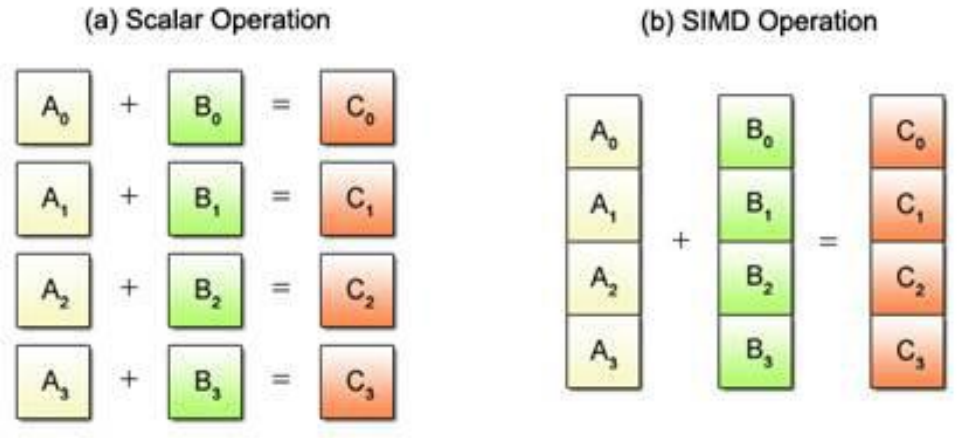


Fig.2.8

El següent inconvenient que planteja és el correcte ordre en la realització de les operacions. El problema consisteix en l'error al precisar si es fa una operació o un altre, provocant fins i tot la pèrdua d'un cicle complet de rellotge.

Exemple:

Taula.2.4

Operació/Cicle	Estat
Suma / 1	Correcte
Resta de cicle 1 / 2	Correcte
Suma cicle 2 amb dades que encara no han entrat /3	Incorrecte
Reinici del procés i els càlculs erronis / 16	Incorrecte

En aquesta taula es pot veure un procés que seria erroni. El problema ha estat que no s'ha realitzat una bona estimació de temps que es trigaria a fer certes operacions. Amb això es vol indicar que si un procés s'executa en la següent iteració, però encara no s'han rebut unes dades que es necessiten per fer aquestes operacions, perdem totes les operacions errònies que s'efectuïn fins que rebem aquestes dades. Un cop rebudes aquestes dades, el procés tornaria a començar per poder realitzar el procés correctament.

2.4. Transferència de dades

En l'arquitectura CBE, existeixen varis tipus de memòria: la memòria principal del PPE i la memòria local de cada SPE. El SPE per tant no pot accedir a la

memòria principal d'una manera totalment directa, igualment que el PPE no pot accedir a les memòries dels SPE's.

Per tant, per accedir a les dades del SPE o PPE, s'ha de transmetre les dades de la memòria local a la principal, o a l'inversa. Aquestes transaccions entre les diferents memòries es realitzen per mitjà de les operacions DMA, que es transmeten a través del bus EIB, i poden ser executades tant pel SPE o PPE.

2.4.1. Transferència iniciada per SPE

Aquest tipus de transferència es genera des de el SPE fins al PPE. És a dir, el SPE envia les dades de la seva memòria local a la memòria principal del processador. En aquest punt, el processador principal, o PPE, pot interpretar les dades llegint-les de la seva pròpia memòria.

- Per poder realitzar la transferència en aquest sentit, cal incloure la llibreria:
  - #include <spu_mfcio.h>

- Per començar s'ha de reservar un tag, i al acabar s'ha d'alliberar. Aquest tag es un identificador unívoc de cada transmissió DMA. Si al reservar-ho tot a anat bé, la funció retorna un enter amb el valor del tag o un error.
  - mfc_tag_reserve()
  - mfc_tag_release(unsigned int tag)

- A continuació ja es poden realitzar les lectures (un bloc de memòria de la principal a la local) i d'escriptura (de la memòria local a la principal). El bloc de memòria local comença a la direcció LSA i el de memòria principal a la direcció EA. També s'ha d'informar quin és el tag que s'ha reservat anteriorment.
  - 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)

- Per últim abans de tancar la transmissió, hem d'assegurar-nos que les transaccions (tags) han finalitzat. Per fer això tenim que realitzar una mascara que ens indicarà quin és el tag que hem de monitoritzar. Mentre que tindrem una altra que s'esperarà fins que tots els tags asignats per la funció anterior acabin. Després d'aquest punt ja podrem fer servir la comanda release.
  - mfc_write_tag_mask(l<<tag)
  - mfc_read_tag_status_all().

2.4.2. Transferència iniciada pel PPE

En aquest cas, ens trobem davant de la situació contraria a l'apartat anterior. El PPE envia dades de la seva memòria principal a la memòria local del procés executat a cada SPE en particular.
Les operacions que es realitzen són molt similars a les iniciades pel SPE. Les tres funciones diferents són:

- 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).

Les dues primeres funcions funcionen exactament igual que les executades pel SPE. És a dir, encara que són cridades pel PPE, fan la mateixa funció que en el SPE, és a dir, el *put* copia de la memòria local a la principal i el *get* el contrari.

La última funció que és una mica diferent substitueix al *write_tag_mask* i *tag_status*. Amb el paràmetre *behavior* es permet posar el valor SPE_TAG_ALL per iniciar a l'espera per que finalitzi la transacció. Com en aquesta instrucció també s'inclou la màscara no es necessària una instrucció addicional per aquesta tasca.

2.4.3. Alineament i mida de les dades

Degut a la forma optimitzada en la que es realitzen les transaccions entre memòria principal i local, les dades a transferir han de complir:

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

Per tant, s'ha de cuidar que al transferir, les dades s'han de completar fins ha adherir fins un múltiple de 16 bytes, fins un màxim de 16 KB.

Pel que fa a l'alineament és una tasca més senzilla, ja que el compilar disposa de la directiva *aligned*. Així doncs, es fa servir:

Int a[16] __attribute__(aligned(16)).

2.4.4. Double buffering:

L' utilització, en un procés de programació paral·lela amb un processador CBEA, de transmissions DMA suposen un alt percentatge de temps d'execució i de recursos. Per tant, millorar la eficiència de transferència DMA és una manera de realitzar aplicacions molt més ràpides. Una tècnica per millorar

l'eficiència de la transferència DMA, mentre es maximitzen les operacions SPE, és el *double buffering*.

El procés estàndard que hem explicat anteriorment seria el següent:

1. Iniciar la transferència DMA (GET) cap al buffer d'entrada.
2. Esperar perquè el pas 1 estigui completat.
3. Processar les dades i emmagatzemar els resultats calculats al buffer de sortida.
4. Iniciar la transferència DMA (PUT) des de el buffer de sortida.
5. Esperar a que el pas 4 estigui completat.

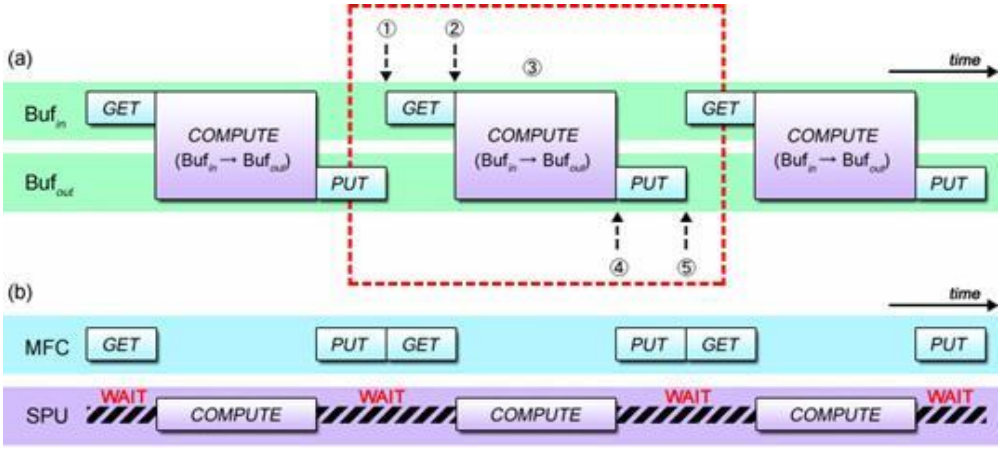


Fig.2.9

Per tant, es pot observar com el MFC no realitza res mentre el SPU esta executant una tasca, o el SPU esta parat mentre la transferència DMA és completa pel MFC. Això impossibilita aprofitar tota la capacitat del SPU.

El mètode de doble buffer utilitza dos buffers separats en paral·lel per entrada i sortida. Així doncs,, mentre un esta usat per computar, l'altre s'està utilitzant per transferència DMA.

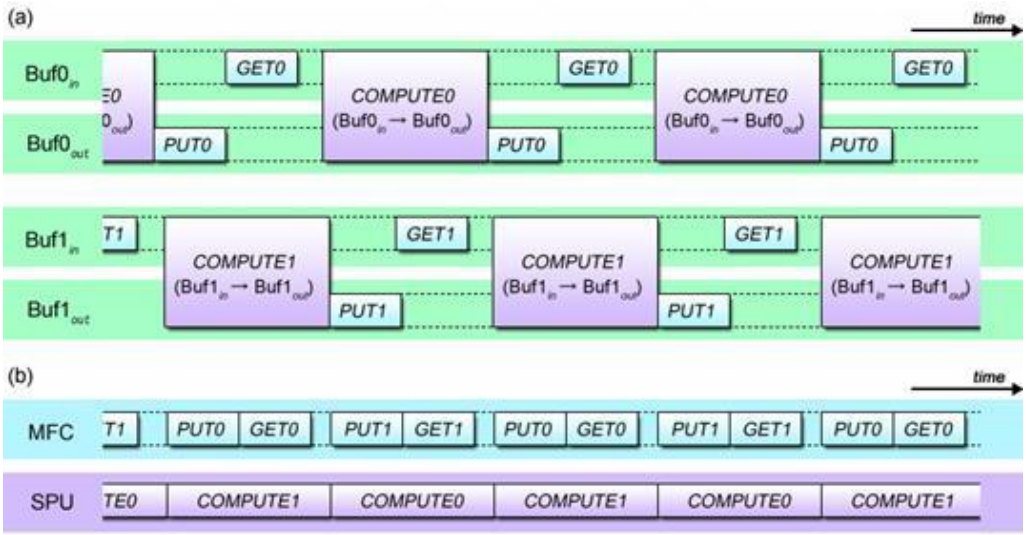


Fig.2.10

Es pot observar com el MFC realitza transferències DMA en paral·lel amb els càlculs del SPU, per tant el SPU no perd temps ha l'espera de la transferència.

Per tant, el procés de *double buffering* és:

1. Iniciar la transferència DMA (GET) cap al buffer 0.
2. Iniciar la transferència DMA (GET) cap al buffer 1.
3. Esperar per la completa transferència DMA (GET) cap al buffer d'entrada 0.
4. Esperar per la completa transferència DMA (PUT) des del buffer de sortida 0.
5. Realitzar les operacions necessàries fent servir les dades del buffer d'entrada 0 i del buffer de sortida 0.
6. Iniciar la transferència DMA (PUT) des del buffer de sortida 0.
7. Iniciar la propera transferència DMA (GET) cap al buffer d'entrada 0.
8. Canvi entre el buffer d'entrada/sortida 0 cap a 1.
9. Iniciar els passos del 3 al 8 fins que totes les dades d'entrada siguin processades.
10. Esperar fins a l'ultima completa transferència DMA (PUT) des del buffer de sortida 0 i 1.

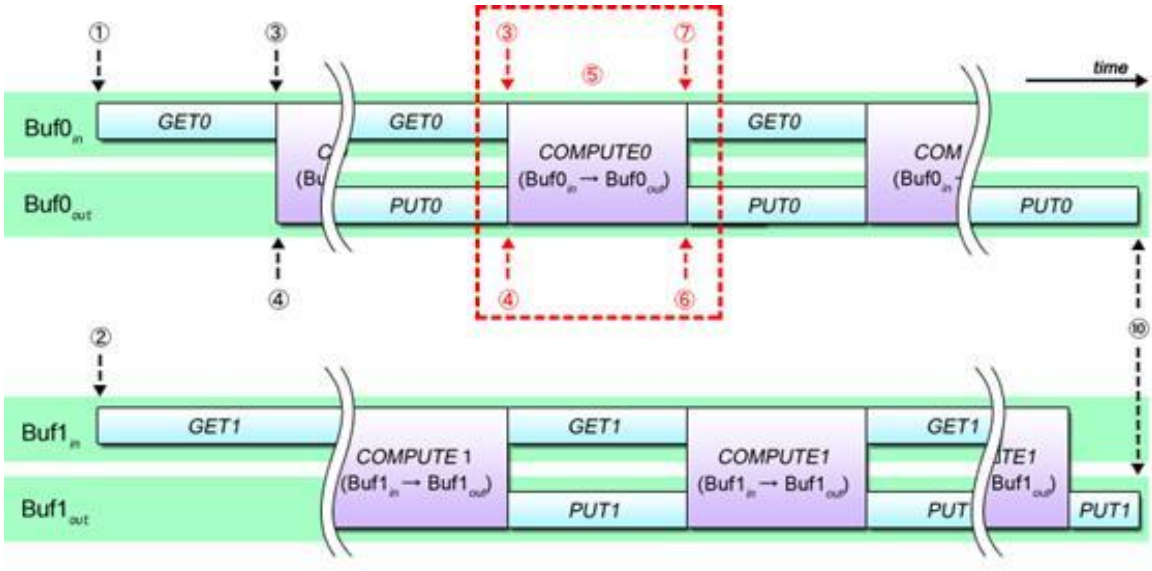


Fig.2.11

El doble buffer esta basat en dos principis bàsics. El primer, la comanda de transferència DMA és utilitzat com buffer d'entrada/sortida des de el començament. El segon, la completa transferència DMA no es verifica fins el moment en que el buffer d'entrada/sortida és utilitzat per fer càlculs.

2.4.5. Mailboxes

Mentre les transferències DMA permeten transferir 16 Kbytes d'informació entre la memòria principal i cada memòria local dels SPE, els mailboxes estan dissenyats per transferir 32 bits d'informació entre el PPE i el SPE.

Així doncs, els mailboxes serveixen per transferir petites quantitats de dades com informació d'estatus i paràmetres. Estructuralment, mailboxes son cues FIFO. El MFC proveeix tres tipus de cues de mailboxes, cadascuna amb un comportament i direcció de transferència de dades diferent.

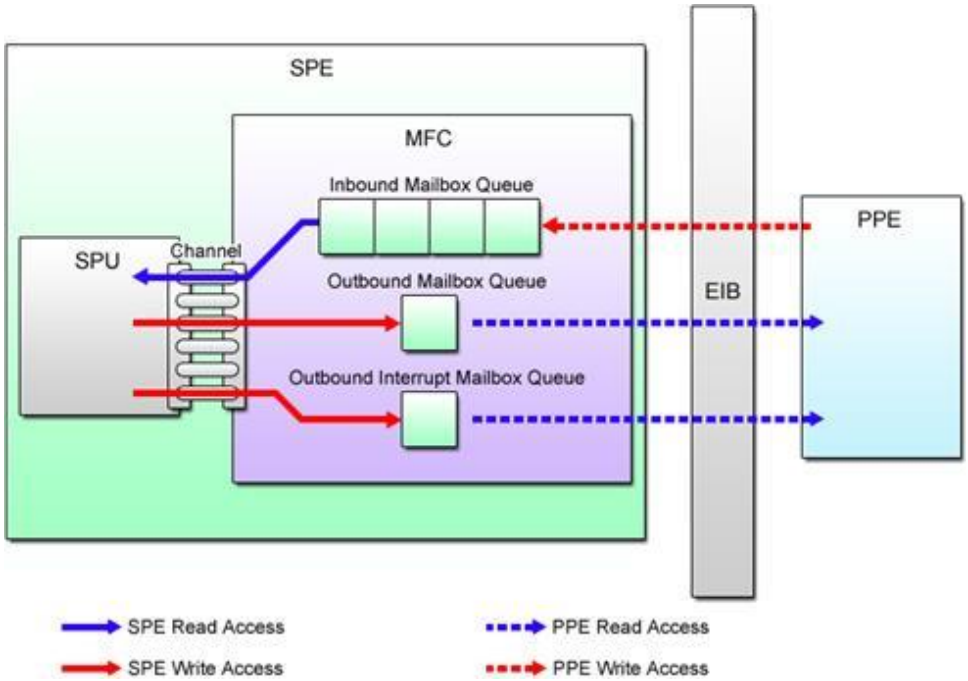


Fig.2.12

1. SPU Inbound Mailbox: s'utilitza per enviar informació des del PPE fins al SPE. Aquest mailbox pot guardar fins 4 missatges de 32 bits al mateix moment. Si el SPE no troba cap missatge quan accedeix a la cua, espera fins que un valor és escrit pel PPE.
2. SPU Outbound Mailbox: s'utilitza per passar dades des del SPE fins al PPE. Aquest mailbox només té capacitat per un missatge de 32 bits. Si el SPU outbound mailbox es troba ple, l'escriptura de la següent dada es suspèn fins que el PPE llegeix la dada des de la cua.
3. SPU Outbound Interrupt Mailbox: Com l'apartat anterior, aquest s'utilitza per enviar informació del SPE al PPE. Quan aquest mailbox és escrit, es genera una notificació d'interrupció al PPE perquè llegeixi la dada.

Als mailboxes es pot accedir tan des de programes dels SPE com del PPE. El programa del SPE utilitza interfases de canals per accedir, mentre que el PPE accedeix via funcions API de mailboxes oferides per la llibreria libspe2.

2.5. Programes SPE Embedded

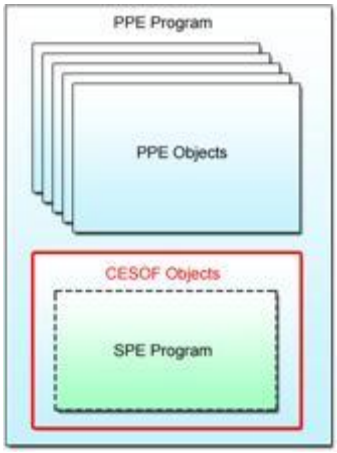


Fig.2.13

Com s'ha explicat anteriorment, generalment el programa en el PPE executa el programa a l'SPE llegint la imatge del programa des de el fitxer executable ELF del mateix SPE. En aquest mètode, el programa de l'SPE i el PPE són compilats en fitxers executable ELF separats.

En canvi, podem trobar un altre mètode on s'utilitza un mateix fitxer executable ELF. En aquest cas, podem crear un programa PPE on introduirem l'anterior programa SPE com un objecte (CESOF).

CAPÍTOL 3. MULTIPROCESSAMENT SOBRE PS3

3.1. Introducció

Amb l'evolució de la tecnologia, hem anat aconseguint cada cop processadors més potents. Amb això aconseguíem realitzar processos més ràpids però igualment lineals amb el consegüent temps d'espera entre l'inici d'un procés i de l'altre. Aquesta situació es va pal·liar amb la creació de processadors amb múltiples nuclis. Aquests multiprocessadors podien realitzar processos diferents a l'hora, maximitzant la potència en conjunt.

El processador de PS3 fa servir aquest concepte però amb 8 processadors en paral·lel.

3.2. Llibreries

Segons el procés que fem servir necessitem unes llibreries particulars.
En les primeres versions del SDK existia la llibreria libspe.h, que ens proporciona la possibilitat de fer servir multithread.y transmissió entre els diferents SPE.

Les dues llibreries més importants utilitzades en aquests processos son:

- Libspe2.h: aquesta llibreria es la millora en qüestió de multithreads de la libspe.h. Mentre la primera versió tenia una única funció per crear el context i generar-lo, la versió següent crea aquesta acció en dos passos. Fent això, han volgut donar més llibertat a l'hora de decidir quan es crea el context i posteriorment quan es vol executar aquest.

- Spu_mfcio.h: en quant a la transmissió entre SPE, aquesta llibreria proporciona les funcions necessàries per realitzar les transmissions DMA.

3.3. Estructura programa multiprocessament

Tots els programes multiprocessament es basen en executar un programa pare, que, arribats a un punt, generen processos fill.

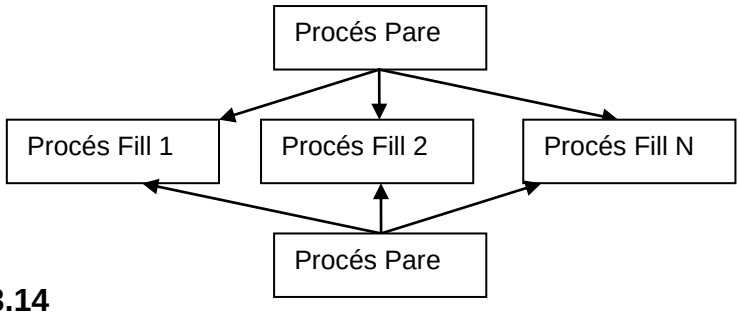


Fig.3.14

Aquest processos fills poden ser diferents, o atacar al mateix programa. La finalitat de cada programa depèn del procés que s'executi, i ja pot acabar primer l'últim com a l'inrevés.
En la majoria de hardware, aquests fills s'executen en un únic processador, però compartint la memòria. En canvi als processadors amb varis nuclis, es pot executar cada un en un nucli diferent.
En els casos de mutithread s'aconsegueix un rendiment més elevat al no tenir que compartir memòria.
Per contra, s'ha de gestionar perfectament l'espera d'un procés a un altre si han de compartir informació entre ells.

3.4. Declaració variables a memòria

Per fer servir la transmissió DMA, hem de reservar l'espai de memòria on es guardaran les dades dels diferents processos.
Per fer aquesta reserva, fem servir la següent funció:

- *posix_memalign* ((void*)(&data_in), ALIGMENT, DATA_BUFFER_SIZE)

Aquesta funció, ens retorna a la dada "data_in", la direcció de memòria on comencen les dades. L'últim camp, ens informa de la longitud que tindrà la reserva de dades. Per últim i més important per les transmissions DMA simultànies esta el camp alienament. Aquest camp ens separa l'inici de la nostra reserva, amb l'inici de les posteriors reserves.

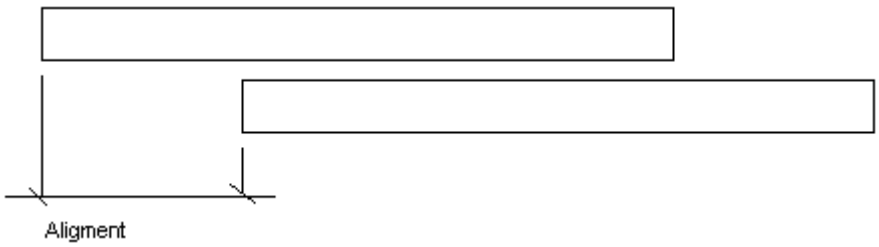


Fig.3.15

Es tant important en una transmissió DMA, perquè si l'alineament no es suficientment gran, podem estar llegint la mateixa adreça de memòria (que no les mateixes dades) a la vegada. Això provocaria una "violació del segment" explicat en apartats posteriors.

En el nostre cas, hem seleccionat una alineació de 65536 per assegurar que no es produeix violació del segment. Es podria haver seleccionat 32768 que es la longitud límit múltiple de 2 del nostre array, però aquest punt es la fi d'un array y el començament del següent.

En aquest punt es on trobem la decisió de tenir un programa més eficient en quant a memòria, o un procés més estable. En aquesta situació, si escollim la primera opció, obtindrem una separació entre dades suficient per que no ocupin una mateixa adreça de memòria. En canvi, si escollim la segona opció ens

trobarem que ocuparem un espai de memòria molt més petit, però tenim un risc elevat de poder intentar llegir el mateix espai de memòria a l'hora.
Una opció intermitja entre les dues exposades, seria que l'alineament tingués el mateix valor que la mida màxima de les nostres dades. Així només es podria donar que estiguéssim llegint l'ultima posició d'una de les dades i la primera de la següent. Els errors que es podrien produir s'explicaran en apartats posteriors.

3.5. Funcions per multiprocessament en CBEA

Per realitzar un programa multiprocessament amb la programació per CBEA, hem de seguir els següents passos:

- Comprovació que existeixen SPE lliures per generar un procés.
    - *spe_cpu_info_get*
- Creació del context per preparar el hardware per la recepció del programa.
    - *spe_context_ptr_t spe_context_create(unsigned int flags, spe_gang_context_ptr_t gang)*

  Si hem tingut exit la funció ens retorna un punter amb el contexte de SPE creat. En canvi, si retorna NULL indica que ha hagut un error..
- Carrega un programa principal de SPE, que ha estat mapejat a la memòria a la adreça apuntada per *program,* a la memòria local del SPE identificat per contexte *spe*. Per tant per executar diferents programes, hem de generar anteriorment un contexte amb aquests programes i cridar aquesta funció canviant el segon paràmetre que ens indica la direcció de memòria on esta situat el programa. Aquest SPE no es podrà fer servir per executar un altre programa fins que no s'alliberi del procés actual.
    - *spe_program_load(spe_context_ptr_t spe, spe_program_handle_t *program)*
- Creació dels diferents threads per aconseguir el processament en paral·lel. Sense aquesta comanda, es generarien fills, un darrera l'altre executant-se seqüencialment.
    - *(pthread_create (&data.pthread, NULL, &ppu_pthread_function, &data)*
        - El tercer paràmetre d'aquesta funció es una subrutina en la qual es crida a l'execució del contexte. Aquesta crida bloqueja al SPE i espera fins que ha parat, tant per si es una terminació anormal com si es la fi del programa executat per aquest.

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

- Per assegurar la correcta finalització dels diversos processos, es fa servir una funció que espera els SPE per ordre . Si el SPE a finalitzat anteriorment, no es detectat com a tal i la funció ho comprova..
    - *pthread_join(threadid[i], (void **)status)*
- Per últim, alliberem cadascun dels SPE per si cal fer-los servir en un altre procés. També s'alliberen tots els recursos utilitzats durant l'execució del programa.
    - *spe_context_destroy (spe_context_ptr_t spe)*

3.6. Transmissió entre SPE

Com ja hem comentat en apartats anteriors, existeixen diversos mètodes per a transmetre informació entre el PPE i els diversos SPE's (o entre diversos SPE's directament). En el nostre cas, nosaltres hem escollit un sistema de transmissió DMA pels següents motius:

- En el sistema de signals, només es pot enviar un número reduït de dades. Aquest sistema es pot fer servir per indicar entre diferents processos que poden realitzar tasca que fins a aquest moment no podien.
  Per exemple: tenim un SPE1 que esta acabant de realitzar uns càlculs i quans els acaba, avisa a un SPE2 que ja ha finalitzar aquesta tasca. Llavors en aquest moment el SPE2 pot començar a realitzar la lectura de les dades esperades, sense necessitat de llegir el mateix espai de memòria a la vegada (provocant un error de violació del segment).
- El sistema de mailboxes, permet transferir dades entre el PPE i el SPE. A nosaltres aquesta casuística sens queda curta. Això es degut a que necessitem transmetre informació entre SPE sense tenir que passar pel PPE. Si necessitem passar pel PPE, es perdria eficiència esperant que el inputs/outputs del PPE estiguessin lliures.

Els requisits per fer servir el tipus de transmissió DMA són el següents:

- Per cada transmissió DMA podem transmetre un màxim de 16Kb. Si es supera aquesta capacitat, hem de realitzar trossos de la totalitat i fer mini-transmissions. Aquest procés es pot fer amb el que s'anomena DMA list.
  En el nostre cas, hem escollit un mida de transmissió de 128 unitats d'un array de 24 bits cada una, es a dir, 3072 bits. Hem escollit aquest mida, per poder realitzar lectures del mateix fragment de memòria sense tenir problemes de concurrència.

Per tant, obtenim 234 transmissions DMA de 128 unitats i 1 transmissió de 48 unitats.

- Ja hem comentat que la mida màxima de la transmissió per DMA es de 16Kb, encara que hi ha una altra condició per la correcta transmissió pel que fa al mida, i es que, aquests paquets han de ser múltiples de 16, per això nosaltres hem escollit 3072.
- Com ja s'ha comentat en l'apartat anterior, totes les dades (i tipus de dades on es contindran) que intervinguin en les transmissions DMA, han d'estar alineades a 16 bits.

Totes les transmissions DMA al sistema Cell/B.E. han de seguir certes regles. Per la naturalesa dinàmica del processat DMA, un paràmetre a una comanda que no segueix les regles, pot no causar un error durant la compilació. En canvi, durant l'execució, la cua de processat de la comanda MFC es suspesa, i una excepció es enviada cap al PPE.

- **Bus error**

Taula.3.5

Tipus d'error	Error	Descripció
Error mida DMA	Mida transmissió erroni	El mida de transmissió no es 0,1,2,4,8 bytes o múltiples de 16
Error mida DMA	Mida de transmissió és massa gran	El mida de transmissió és més gran de 16Kb
Error mida DMA	Mida de la llista de transmissió és massa gran	Element de la llista té mida més gran de 16Kb.
Error d'alineament DMA	Emmagatzament de l'adreça local	Les adreces origen i destí no estan alineades a menys de 16 bytes o no estan alineades amb múltiples de 16 bytes
Error d'alineament DMA	Emmagatzament de l'adreça principal	Les adreces origen i destí no estan alineades a menys de 16 bytes o no estan alineades amb múltiples de 16 bytes
Error d'alineament DMA	Alineament de la llista d'adreces	La llista DMA no esta emmagatzemada a la memòria local del SPE en un múltiples de 8 bytes.
Error Tag ID	Tag ID invalid	Tag ID no es troba entre 0 i 31 inclosos
Altres	La llista de elements supera el límit de 4GB	Per les aplicacions de 64 bits, la llista de elements no pot superar el 4GB. (No per aplicacions de 32 bits).

Algunes comandes DMA, poden provocar errors que no són detectats pel hardware, causant una violació del segment o fallida. Els errors de segmentació que ocorren al SPE són sempre causats per transmissions DMA cap i/o des de adreces errònies.

- **Violació del segment**

Taula.3.6

Tipus d'error	Descripció
Invalida adreça destí	**Una comanda DMA PUT amb una adreça a la que no es pot accedir.**
Invalida adreça origen	**Una comanda DMA GET amb una adreça a la que no es pot accedir**

3.7. Funcions SIMD per realitzar operacions vectorials

A l'apartat 2.3 ja hem comentat aquestes funcions amb una mica de profunditat teòrica. En el cas que ens pertoca, hem estudiat si es poden introduir al nostre codi per millorar la nostra eficiència.

Primer de tot, per poder fer servir aquestes funcions, es necessiten que les dades siguin del tipus "vector". A més, els components d'aquests tipus de vectors han de ser, generalment, diferents a integer. Les funcions que permeten integers com paràmetres d'entrada són per exemple *absi4* i *negatei4*, Les funcions restants fan servir tipus d'entrada "vector float" o "vector double".

Per tant, en el nostre cas tenim les següents incompatibilitats impedirien fer servir aquest tipus de funcions.

Vector ←→ Arrays
Float ←→ Integer

Per fer servir les funcions SIMD, tenim que reestructurar els arrays perquè es requisit fer les operacions de 4 en 4 elements. Com els nostres arrays no son d'aquesta mida, per fer les operacions amb la millora que ens proporcionen aquestes funcions, hauríem d'adaptar les dues dades a combinar. Agafaríem els quatre primers valors del vector original i copiar-lo a un vector vàlid, el mateix amb el segon vector. Llavors el resultat de l'operació l'hauríem de passar a un vector del format del programa global. Per tant, ens es més fàcil realitzar les operacions individualment, posat que per cada transmissió DMA, hauríem de generar un altre bucle per realitzar l'operació corresponent.

En el nostre petit programa no obtindríem eficiència de convertir les dades. En canvi en una situació on es fessin servir per realitzar varies operacions amb les mateixes dades, es podria "perdre" aquesta petita quantitat de temps per realitzar la conversió.

El que si es cert, es que si es disposa de la possibilitat de treballar directament amb tipus de dades vector i que no siguin integers, la velocitat de processament de les operacions és molt més superior amb funcions vectorials (SIMD) que amb les funcions clàssiques.

CAPÍTOL 4. RESUM

En aquest projecte hem desenvolupat un codi basat en l'arquitectura multiprocessador CBEA. Hem realitzat un programa que genera processos e intercanvia informació entre ells.

L'estructura bàsica del programa realitzat es la següent:

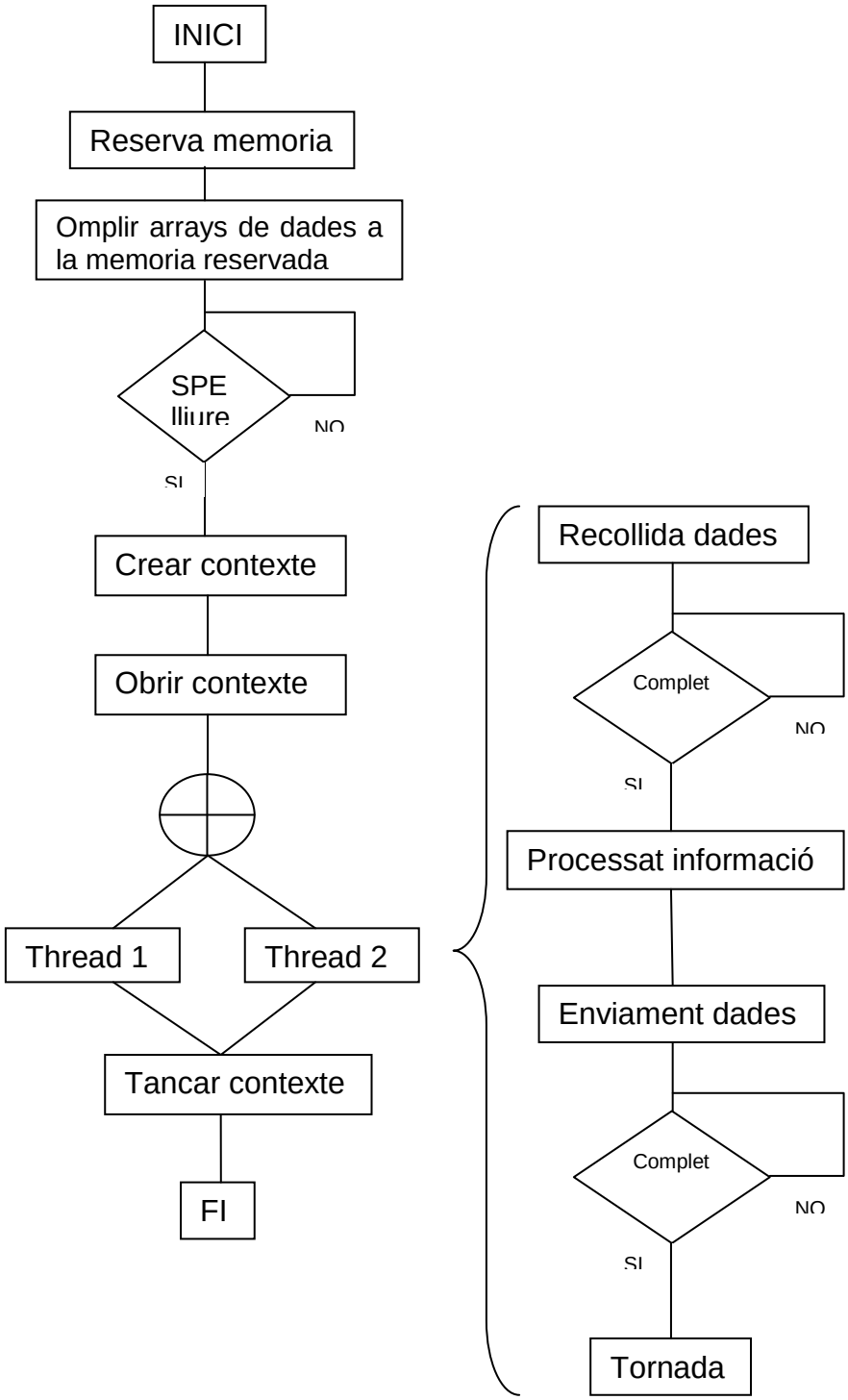


Fig.3.16

L’arquitectura es basa en un fitxer que conte el codi del procés principal, i en aquest cas, dos fitxers que contenen cadascú un programa que realitzen processos diferents. Al fitxer principal es creen les diverses variables y es reserven els espais a memòria on es guardarà la informació. Aquest pas es molt important que estigui definit correctament, ja que es podrien generar problemes de lectura i escriptura simultanis provocats pel paral·lelisme. A més, sense aquesta bona definició no podríem accedir a la informació des de els diversos subprocessos. Els altres dos fitxers que tenim són els subprocessos que es generaran quan el principal els cridi. S’executen simultàniament pel que s’ha tingut en compte la no concurrència de memòria, que ens ha provocat errors d’execució.

En un dels programes es rep l’ informació i es realitza una operació que es retorna al procés principal. Uns moments més tard del començament del primer subprocés, el segon rep informació del procés principal i un cop finalitzada aquesta recepció, comença una altra lectura per aconseguir les dades retornades pel primer subprocés al principal. A continuació aquest segon, realitza una operació amb aquestes dades i les retorna al principal. Finalment, mostrem per pantalla algunes dades per demostrar que s’ha realitzat correctament la transferència de dades.

Aquest programa que s’ha realitzat, es la base de qualsevol tipus de programa que necessiti executar processos paral·lels totalment diferents. Entre la recepció de dades i la transmissió de dades entre sistemes, es pot introduir qualsevol codi per manipular aquesta informació i transmetre-la perquè un altre procés posterior (o simultani) l’utilitzi.

En un començament, es va realitzar un procés que executava dos processos paral·lels però sense intercanvi d’informació. Quan es va intentar incloure la comunicació entre PPE y SPE de la manera estàndard no es va poder realitzar. Llavors si es feia servir la funció que arrenca el thread, amb la direcció de memòria reservada anteriorment, no funcionava correctament. Així doncs, es va intentar utilitzar una estructura on s’indicava la direcció inici de cada subprocés. El problema d’aquesta idea es que si intentàvem llegir-la a la vegada obteníem problemes de concurrència. La sol-lució va ser definir la direcció de les adreces de memòria de manera estàtica. Això es va aconseguir un cop ja es tenen definits completament tots els arrays y dades del programa. En aquest moment, fem servir les direccions que ens ha retornat la funció de reserva de dades (que són sempre les mateixes si no es modifiquen els tamanys dels arrays) per saber on estan situades les dades dels nostres processos.

Per realitzar l’estructura segons l’ordre d’execució del nostre programa, hem de tenir en compte la quantitat de processos que es poden realitzar simultàniament. El sistema disposa de fàbrica de buit processadors, però només son servibles sis. Això es degut a que un d’ells està reservat pel sistema gràfic de la PS3, i l’altre de fàbrica ja el tenim inhabilitat per assegurar que es tenen sis processadors útils. Per tant, els podem fer servir per realitzar sis processos simultanis o, potenciar algun fil que estiguem realitzant. Per exemple, si estem realitzant 5 processos, i hem d’esperar dades d’aquests per

continuar, podem dividir un dels processos per la meitat i així es podria executar de manera paral·lela aconseguint el doble de velocitat en aquest procés.

Per arribar a aquest programa final, ens hem trobat amb una gran quantitat de problemes:

- Concurrència de memòries al llegir i escriure. Per sol·lucionar aquesta situació el que es va implementar va ser la separació completa de dades de memòria compartida. En la declaració de memòria es pot declarar un paràmetre que indica el següent paràmetre de memòria que es podrà fer servir. Per tant, es va impedir que la següent dada de memòria possible estigués continguda a una informació anterior.
- Dades de longitud incorrectes. En aquest cas, es van fer servir (per requeriments de disseny), dades de memòria que no eren de longitud potencia de 2. Per tant, es va modificar la longitud de les dades perquè fossin potencia de 2, però superiors a les dades prèvies de disseny.
- Transmissió de dades. La transmissió de dades ha de ser múltiple de dos i esta limitat per una màxima quantitat a transmetre. Així doncs, com la nostra informació era molt superior a la permesa per transmetre de cop, es va dividir ens petits trams a transmetre.
- Ordre de execució. Ja que l'execució no es lineal, posat que es poden executar processos a la vegada, s'ha de tenir compte amb l'ordre en el qual s'executen per tenir les dades en els moments indicats. Es va tenir que fer testeitx de temps per saber si quan s'executés el segon procés ja estaria el primer finalitzat. Aquest testeig va permetre definir la quantitat de dades necessàries per tenir informació rebuda(encara que no fos el màxim) i no perdre temps d'execució.
- Memòria màxima per SPE. En un començament, es van intentar definir un array d'entrada i un de sortida. Això va ser perquè així podríem rebre informació i transmetre-la, i a més poder realitzar operacions amb totes les dades de l'array. Quan es creava una quantitat d'arrays massa gran, es creava un error posat que es sobrepassava la memòria permesa per cada processador.

Definint aquesta longitud de paràmetres obtenim una certa seguretat, però perdem eficiència. Això es degut, a que si podem realitzar una transmissió DMA màxima de 16Kb, nosaltres utilitzem uns 3Kb, per tant:

$$\text{Eficiència} \approx (3/16) * 100 \approx 18.75\ \%$$

No es un resultat molt eficient, però quan existia coincidència de memòria podíem controlar d'una manera més clara que no tinguéssim els punters del arrays al mateix lloc. La sol-lució trobada, també planteja la poca eficiència del muntatge.

Si observem el paràmetre que separa una zona de memòria de la resta, nosaltres no aprofitem res d'aquest paràmetre. Si tenim una separació màxima de 2^x (per assegurar la no coincidència de memòria), nosaltres fem servir 2^{x+1}, amb el que no estem al límit, sinó que estem desaprofitant memòria.

Així doncs en quant a eficiència podem concloure que estem a les següents gràfiques:

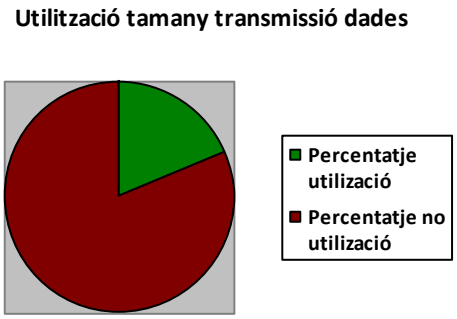


Fig.3.17

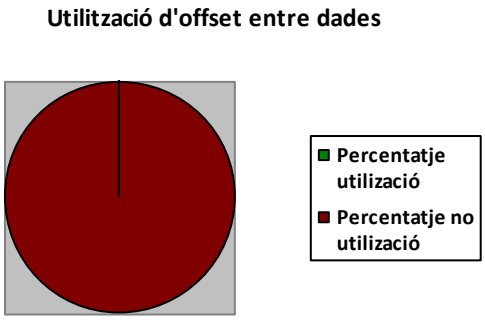


Fig.3.18

CAPÍTOL 5. RESULTATS i CONCLUSIONS

5.1. Resultats

Durant el procés de creació del programa resultant d'aquest treball, es van tenir que realitzar decisions necessàries per tal de no obtenir els errors comentats anteriorment.

Quan realitzem la reserva de dades hem de tenir en compte dos aspectes importants. El primer és que la longitud de les dades ha de ser potencia de 2. En el nostre cas nosaltres hem escollit 32768 ja que necessitem un array de 30000. Amb aquesta restricció pot donar-se el cas que obtenim dades molt superiors a les que necessitem, incrementant la memòria requerida i sobrant espai. Això al PPE no tindria molta importància, però al SPE si. Al SPE tenim una memòria local molt més limitada i si no ajustem les dades el màxim ens quedaria espai inútil. La segona restricció es la separació entre els inicis dels diferents arrays. A la crida de posix:memalign tenim un paràmetre que ens indica aquesta separació. Si la separació es 0, les dades començaran separades l'equivalent a una transmissió DMA, tenint el risc de que al ocupar mateixos espais de memòria, estem llegint la mateixa direcció de memòria i provocant un error. Llavors quant més ens separem, en principi tindrem menys perill de concurrència de memòria. El límit màxim amb menys risc seria l'equivalent a la longitud de l'array. En aquest límit únicament coincidiria si s'està llegint la primera dada d'un array y la última de l'anterior, minimitzant al màxim el risc.

Com fem servir aquesta magnitud de dades tan elevades, hem de realitzar les transmissions DMA fragmentades ja que només es poden transmetre un màxim de 16Kb. Com nosaltres utilitzem dades de tipus integer, cada element té un valor de 24 bits, amb el que hem escollit cada transmissió de 128 unitats (equivalent a 3072 bits per transmissió DMA). Aquest número no es el límit permès de 16Kb, però aconseguim una concurrència de memòria entre diverses transmissions DMA menys probable. Així doncs nosaltres obtenim 234 transmissions de dades màximes (128 unitats de dades) i una 1 transmissió de 48 unitats. Encara que obtenim un control més gran amb aquesta longitud de transmissió DMA, hem de realitzar una quantitat més elevada de iteracions en el bucle. L'augment d'iteracions al bucle ens provoca la repetició de totes les funcions de transmissió un número de vegades més elevat que si escollíssim dades properes al màxim de 16Kb.
Ja s'ha comentat anteriorment el problema que obtenim en el nostre cas amb les funcions SIMD. Aquest tipus de funcions admeten majoritàriament tipus de dades float i en el nostre cas tenim integer. Per solucionar aquest problema podríem copiar les dades temporalment a un altre array de floats sense perdre, en principi, informació. Si el cas fos a l'inrevés no es podria realitzar perquè els floats contenen decimals que al integer desapareixerien. Aquest seria en cert punt un problema lleu pel canvi del format de dades, però a més, hem de canviar el tipus de contenidor d'aquestes dades. En el nostre cas nosaltres fem servir un tipus array i per fer servir aquestes funcions cal que siguin de tipus vector. En aquesta conversió si que perdem part de l' informació, però no es el gran problema de no poder-les fer servir. Arribats a aquest punt s'ha d'escogir un altre cop entre eficiència o comoditat. És a dir, si fem aquestes conversions,

hem de realitzar bucles per igualar els tipus de dades amb el que tindrem iteracions extres que fan perdre temps de computació. En canvi amb aquest procés guanyarem temps amb l' utilització de les funcions SIMD que fan les operacions de 4 en 4.

A més, si tenim processadors lliures, podem tornar a la base de l'arquitectura CBEA: potencia un procés. El que faríem seria dividir l' informació que s'envia a un procés, i enviar-la a tants com tinguem lliures. Aquesta situació la podem trobar, per exemple, quan fins que no s'acabi un procés no podem començar un altre. Llavors faríem servir els processadors lliures per poder acabar abans i enviar l' informació al següent procés més aviat.

5.2. Conclusions

Finalitzat l'estudi sobre la possible aplicació d'una arquitectura Cell Broadband Engine , i realitzat el codi base per poder realitzar qualsevol programa que necessiti transmissió entre processos, ens fem una pregunta: "Es viable substituir processadors clàssics pels d'última generació de multiprocessament?".La nostra conclusió seria que es totalment viable substituir aquest tipus de processadors ja que ens proporcionen certs avantatges.

Actualment els processos que creem són multithreat però no multiprocessador. Amb això volem dir que encara que podem generar diversos processos simultanis, aquest s'executen en el mateix processador. Aquest cas, ens genera una repartició dels recursos físics per cada un dels subprocessos generats. Això es podria sol·lucionar amb una arquitectura de multiprocessament. En aquest cas aconseguiríem que s'executessin els mateixos processos esmentats anteriorment, però no es repartirien els recursos del processador, perquè cada un tindrà els seus recursos propis. Amb això aconseguim una millor eficiència en l'execució dels programes.

Per aquest mateix cas de multiprocessadors, ens trobem certes barreres a superar. En primer lloc si necessitem que els processos siguin totalment paral·lels però tenim informació dependent en el flux de processos, hem d'aconseguir una sincronia total. Si aquesta sincronia no existís, el que provocaria seria falta d'informació en algunes operacions i, per tant, resultats no esperats en els processos.
La següent barrera seria la compartició de l' informació. En els actuals processos multithreat com tots s'executen al mateix processador, podem fer servir les dades entre ells ja que fan servir els mateixos recursos físics. En canvi, al multiprocessador que fem servir, com executem cada fil en un processador amb els seus propis recursos, si necessitem informació entre ells, hem de fer lectures a dades de memòria contingudes al processador principal. Aquestes lectures ens provoquen una demora de temps i una possible concurrència de lectura/escriptura que pot acabar en pèrdua d'eficiència o fins i tot mal funcionament del programa.
Aquests petits inconvenients, son fàcilment sol·lucionables per un programador. El programador es el que decideix en quin moment s'executa un procés o un altre, com han de ser de llargues les transmissions de dades entre processos, i una quantitat de comandes particulars que fan més eficient certes operacions.

Per tant, si la persona que crea el programa, coneix a la perfecció el procés que vol crear, podrà aconseguir incorporar les particularitats que té el nostre multiprocessador aconseguint els resultats desitjats en menys temps que els processadors convencionals.

El que no s'ha pogut acabar de fer, es realitzat una simulació d'un procés de comunicació per Wimax. S'hauria d'haver inserit als processos fills el codi que simulen els diferents mòduls de Wimax.

Així doncs, la continuació hauria de ser modificar aquest codi per poder implantar els diferents threads que generessin els mòduls del procés. A més, hauríem d'observar l'ordre d'execució i de transmissió per saber el moment indicat d'execució de cada procés. També es tindria que mirar si es tenen processadors lliures per poder potenciar algun procés que tinguem executant-se. A més, posat que no aprofitem l'eficiència de memòria que ens proporciona aquest sistema, hauríem d'intentar reduir al màxim la separació i la fragmentació dels arrays.

Annexes.

Instal·lació software a PS3

Actualment, existeixen diferents distribucions que es poden instal-lar a la PS3, com per exemple Yellow Dog Linux, Helios Linux, Fedora Core 6, Ubuntu, Gentoo o Debian.

Després de tenir la ISO de qualsevol d'aquestes distribucions i la del Linux add-on, hem de generar el carregador d'arrencada. Aquest l'obtenim del directorio kboot del Linux add-on (fitxer otheros.bld). Llavors a un pendrive creem un directori anomenat PS3 i dintre un altre, amb nom otheros, on copiarem el fitxer mencionat anteriorment.

Nota: En aquest punt es important ressaltar que es necessari formatejar el disc dur de la PS3, pel que es recomana fer una copia de seguretat.

Per formatejar el sistema hem d'accedir al menú "Ajustes/Ajustes del sistema/Herramientas de formateo/Formatear disco duro". En aquest cas genera dues particions, una de les quals es de 10 GB.

Per instal·lar el gestor d'arrencada, es necessari connectar el pendrive i accedir al menú "Ajustes/Ajustes del sistema/Instalar otro sistema operativo". Llavors el propi sistema localitzarà l'arxiu otheros.

El següent pas es canviar el sistema operatiu per defecte i reiniciar. Per fer-ho, s'ha d'accedir a "Ajustes/Ajustes del sistema/Sistema predeterminado" i escollir la opció "Otro sistema operativo". En aquest punt depèn de cada distribució. Per exemple a Fedora Core 6, després de l'ordre kboot, hem d'introduir la comanda install-fc sda. Llavors demana el DVD d'instal·lació i en algun cas el Linux add-on CD.

Una vegada instal-lat el sistema operatiu, la comanda *boot-game-is* permet arrencar PS3 amb el sistema operatiu dels jocs.

Instal·lació software i SDK a CPU

Per instal·lar el SDK, en un sistema operatiu Fedora, s'ha de seguir els següents passos:

1. instal·lar el script per realitzar la instal·lació:

   # rpm -ivh cell-install-3.1.0.0.0.noarch.rpm

2. Fent servir el script de l'apartat anterior, instal·lar el SDK. Opcions:

   a. Si s'està instal·lant des de una imatge iso:

 # /opt/cell/cellsdk --iso /tmp/cellsdkiso install

b. Si s’està instal·lant des de un servidor local:

 # /opt/cell/cellsdk install

c. Si s’està instal·lant des de un CD, munta el CD al directori /media i executa'l:

 # /opt/cell/cellsdk install

3. Llavors acceptar les llicencies i l’instal·lador YUM executarà tots els paquets RPM.

Un cop instal·lat tots els elements bàsics, hem de impedir que es pugui actualitzar el sistema i ens retiri tots els canvis que l'hi hem realitzar al sistema. Per un sistema Fedora, hem d'afegir certes línies al fitxer */etc/yum.conf* (la línia d'exclusió s'ha de trobar en una mateixa línia, no s'ha de separar:

exclude=blas blas-devel elfspe2 kernel lapack lapack-devel libspe2 libspe2-debuginfo libspe2-devel oprofile oprofile-devel

La instal·lació d'aquest script canvia les versions dels RPM blas, lapack i elfspe2, libspe2 i oprofile amb els RPM localitzats al directori /tmp/cellsdk/openSrc.

S'ha de tenir en compte llavors, que si es vol actualitzar a una versió diferent de la que es té, s'haurà de realitzar manualment.

Codi generat per multiprocessament

El primer procés que es realitzarà es el més simple possible. Crearem diferents processos paral·lels que s’executaran a diferents SPE. El que podrem observar es com cada SPE executa el mateix procés, mostrant per pantalla la direcció de memòria de cada un d’ells.
Al SPU, es posaria el següent codi:

```
#define MAX_SPU_THREADS      16


void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  ctx = *((spe_context_ptr_t *)arg);
  if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}
```

```
int main()
{
  int i, spu_threads;
  spe_context_ptr_t ctxs[MAX_SPU_THREADS];
  pthread_t threads[MAX_SPU_THREADS];

  /* Determine the number of SPE threads to create.
   */
  spu_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
  if (spu_threads > MAX_SPU_THREADS) spu_threads = MAX_SPU_THREADS;

  /* Create several SPE-threads to execute 'simple_spu'.
   */
  for(i=0; i<spu_threads; i++) {
    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      exit (1);
    }
    /* Load program into context */
    if (spe_program_load (ctxs[i], &simple_spu)) {
      perror ("Failed loading program");
      exit (1);
    }
    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL, &ppu_pthread_function,
&ctxs[i]))  {
      perror ("Failed creating thread");
      exit (1);
    }
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<spu_threads; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }

    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nThe program has successfully executed.\n");

  return (0);
}
```

Amb això aconseguim crear un context on s'executarà un procés fill com en la programació paral·lela comú. La diferència la trobem en que aquest nou procés, s'executarà en un dels SPE. Amb aquest fet, aconseguirem que el rendiment del nostre programa augmenti, ja que la càrrega de processat es reparteix entre els SPEs.
Pel que fa al codi del SPE:

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

```
int main(unsigned long long id, int i)
{
  printf("Hello Cell (0x%llx)\n", id);
  printf("posicion : %d", &i);
  return 0;
}
```

En aquest cas hem escollit el mateix programa en els diferents SPE, però es podria haver executat diferents processos.
Per poder executar diferents processos, hem de crear diferents carpetes, cadascuna amb noms diferents. També s'hauria de posar a cada crida de cada procés una condició per diferenciar-los. Això es podria fer fàcilment, diferenciant els processos segons la variable i. Llavors, cada crida seria diferent depenent del programa fill que executi.
Per últim, al Makefile, s'hauria d'indicar que existeixen diferents processos pels diferents SPEs. Als codis següents podem veure en blau com s'afegirien:

- Makefile per un mateix programa fill:

```
#######################################################################
##
#                  Subdirectories
#######################################################################
##

DIRS        :=    spu


#######################################################################
##
#                        Target
#######################################################################
##

PROGRAM_ppu :=    simple


#######################################################################
##
#                        Local Defines
#######################################################################
##

IMPORTS         = spu/lib_simple_spu.a -lspe2 -lpthread

INSTALL_DIR = $(EXP_SDKBIN)/tutorial
INSTALL_FILES     = $(PROGRAM_ppu)

#######################################################################
##
#                  buildutils/make.footer
#######################################################################
##

ifdef CELL_TOP
      include $(CELL_TOP)/buildutils/make.footer
else
      include ../../../buildutils/make.footer
endif
```

Makefile per diferents programes fill:

```
#######################################################################
##
#                   Subdirectories
#######################################################################
##

DIRS        :=    spu spu2 spu3 … spun


#######################################################################
##
#                         Target
#######################################################################
##

PROGRAM_ppu :=    simple


#######################################################################
##
#                         Local Defines
#######################################################################
##

IMPORTS         = spu/lib_simple_spu.a -lspe2 -lpthread
spu2/lib_simple_spu.a -lspe2 -lpthread spu3/lib_simple_spu.a -lspe2 -
lpthread … spun/lib_simple_spu.a -lspe2 -lpthread


INSTALL_DIR = $(EXP_SDKBIN)/tutorial
INSTALL_FILES     = $(PROGRAM_ppu)

#######################################################################
##
#                   buildutils/make.footer
#######################################################################
##
```

Si volem fer que aquest tipus de programa comparteixin informació un amb els altres i realitzar operacions amb dades d'altres SPE, hem de fer servir transmissions DMA (apartats 2.4 y 3.5). Aquest tipus de programa es el que hem generat en aquest treball i es el següent.

- PPE.

```
#include <sched.h>
#include <libspe2.h>
#include <pthread.h>
#include <dirent.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <sys/wait.h>
#include <string.h>
#include <simple_dma.h>
```

```
#define MAX_SPU_THREADS 16

/* we allocate one control block, to correspond to one SPE */
control_block cb1 __attribute__ ((aligned (ALIGMENT)));
control_block cb2 __attribute__ ((aligned (ALIGMENT)));

/* this is the pointer to the SPE code, to be used at thread creation
time */
extern spe_program_handle_t simple_dma_spu1;
extern spe_program_handle_t simple_dma_spu2;

/* this is the handle which will be returned by "spe_context_create."
*/
spe_context_ptr_t speid1;
spe_context_ptr_t speid2;

/* this variable is the SPU entry point address which is initially set
to the default */
unsigned int entry = SPE_DEFAULT_ENTRY;
unsigned int entry2 = SPE_DEFAULT_ENTRY;

/* this variable is used to return data regarding an abnormal return
from the SPE */
spe_stop_info_t stop_info;

/* here is the variable to hold the address returned by the malloc()
call. */
int *data_in;
int *data_out;
int *data2_in;
int *data2_out;
int *data_blank_in;
int *data_blank_out;
int *data_blank_inter;

/*Estructura para thread*/
typedef struct {
    spe_context_ptr_t   spe;
    control_block       cb;
} thread_arg_t;

/*Funcion llamada contexto en thread*/
void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

     thread_arg_t *argt = (thread_arg_t *) arg;
     spe_stop_info_t stop_info;

  if (spe_context_run(argt -> spe, &entry, 0, &argt->cb, NULL,
&stop_info) < 0) {
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}

int main()
{
```

```
int i, spu_threads, rc;
  spe_context_ptr_t ctxs[MAX_SPU_THREADS];
  pthread_t threads[MAX_SPU_THREADS];
  thread_arg_t *arg;

  /* Determine the number of SPE threads to create.
   */
  spu_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
  if (spu_threads > MAX_SPU_THREADS) spu_threads = MAX_SPU_THREADS;

  /* Create several SPE-threads to execute 'simple_spu'.
   */
/*********************************************************************
*******/
  /* Here is the malloc call a data array aligned to a cacheline
boundary
     for efficient transfer.
   */

/****SPU1*****/
  rc = posix_memalign ((void*)(&data_in), ALIGMENT, DATA_BUFFER_SIZE);
  if (rc != 0)
  {
    fprintf (stderr, "Failed allocating space for data array\n");
    exit (1);
  }

  printf("Address being sent in control block1in: %p\n", data_in);

  cb1.addr_in = (unsigned long long)((uintptr_t)data_in);

/****SPU2*****/
  rc = posix_memalign ((void*)(&data2_in), ALIGMENT,
DATA_BUFFER_SIZE);
  if (rc != 0)
  {
    fprintf (stderr, "Failed allocating space for data array\n");
    exit (1);
  }

  printf("Address being sent in control block2in: %p\n", data2_in);

  cb2.addr_in = (unsigned long long)((uintptr_t)data_out);

/******************Completamos los primeros valores para hacer
calculos****/
for(i=0;i<6000;i++)
{
      data_in[i]=i;
      data2_in[i]=i+2;
}

/*********************************************************************
*******/

  for(i=0; i<2; i++) {

/******Para los diferentes SPE disponemos de diferentes
llamadas*************/
  if(i==0)
  {
```

```
    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      exit (1);
    }

    /* Load program into context */
    if (spe_program_load (ctxs[i], &simple_dma_spu1)) {
      perror ("Failed loading program");
      exit (1);
    }

   arg[i].spe = ctxs[i];
//   arg[i].cb = cb1;

    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL, &ppu_pthread_function,
&arg[i]))  {
      perror ("Failed creating thread");
      exit (1);
    }
  }
  if(i==1)
  {

    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      exit (1);
    }

    /* Load program into context */
    if (spe_program_load (ctxs[i], &simple_dma_spu2)) {
      perror ("Failed loading program");
      exit (1);
    }

   arg[i].spe = ctxs[i];
//   arg[i].cb = cb2;

    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL, &ppu_pthread_function,
&arg[i]))  {
      perror ("Failed creating thread");
      exit (1);
    }
  }
/******End of para los diferentes
SPE*********************************************/

  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<2; i++) {

    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }
```

```
    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nposicion1:%d\n", data_in[15]);
  printf("\nposicion1:%d\n", data_in[30]);
  printf("\nposicion1:%d\n", data_in[0]);
  printf("\nposicion2:%d\n", data2_in[0]);
  printf("\nposicion2:%d\n", data2_in[10]);
  printf("\nposicion2:%d\n", data2_in[25]);

  printf("\nThe program has successfully executed.\n");

  printf("PASSED\n");

  return 0;
}
```

- SPE1

```
#include <spu_mfcio.h>
#include <stdio.h>
#include <simple_dma.h>

#include <sched.h>
#include <pthread.h>
#include <dirent.h>
#include <stdint.h>
#include <errno.h>
#include <sys/wait.h>
#include <string.h>

/* Here's the local copy of the control block, to be filled by the DMA
*/
volatile control_block cb1 __attribute__ ((aligned (ALIGMENT)));

/* Here's the local copy of the data array, to be filled by the DMA */
int data_out[DATA_BUFFER_ENTRIES] __attribute__ ((aligned
(ALIGMENT)));
int data_part[DATA_BUFFER_DMA] __attribute__ ((aligned (ALIGMENT)));

int main(unsigned long long speid __attribute__ ((unused)),
       unsigned long long argp,
       unsigned long long envp __attribute__ ((unused)))
{

  int i,rc,j,z;
  unsigned int tag_id;
  unsigned long long addr;

  /* Reserve a tag for application usage */
  if ((tag_id = mfc_tag_reserve()) == MFC_TAG_INVALID) {
    printf("ERROR: unable to reserve a tag\n");
    return 1;
  }

  /* Here is the actual DMA call */
```

```
  /* the first parameter is the address in local store to place the
data */
  /* the second parameter holds the main memory address
*/
  /* the third parameter holds the number of bytes to DMA
*/
  /* the fourth parameter identifies a "tag" to associate with this
DMA  */
  /* (this should be a number between 0 and 31, inclusive)
*/
  /* the last two parameters are only useful if you've implemented
your  */
  /* own cache replacement management policy.  Otherwise set them to
0.  */
//  mfc_get(&cb1, argp, sizeof(cb1), tag_id, 0, 0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
*/
  mfc_write_tag_mask(1<<tag_id);

  /* Now, issue the read and wait to guarantee DMA completion before
we  */
  /* continue. */
  mfc_read_tag_status_all();

  cb1.addr_in = 0x10040000;
  printf("Address received through control block1in = 0x%llx\n",
cb1.addr_in);

for(i=0;i<234;i++)
{
  /* DMA the data from system memory to our local store buffer. */
  mfc_get(data_part, cb1.addr_in +(sizeof(int)*DATA_BUFFER_DMA*i) ,
sizeof(int)*DATA_BUFFER_DMA, tag_id, 0, 0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
  mfc_write_tag_mask(1<<tag_id);

  /* Wait for the data array DMA to complete. */
  mfc_read_tag_status_all();

/*Transferimos de la fracción de datos que hemos leido al array
principal*/
  z=0;
  for(j=i*DATA_BUFFER_DMA;j<(DATA_BUFFER_DMA+DATA_BUFFER_DMA*i);j++)
  {
        data_out[j] = data_part[z];
        z++;
  }

}

  cb1.addr_out = 0x10040000;
```

```
  printf("Address received through control block1out = 0x%llx\n",
cb1.addr_out);

//*****************************************************************
*****
// Estaremos leyendo, hasta que el array venga informado de otro SPE

  cb1.addr_in = 0x10050000;
for(i=0;i<234;i++)
{
  /* DMA the data from system memory to our local store buffer. */
  mfc_get(data_part, cb1.addr_in +(sizeof(int)*DATA_BUFFER_DMA*i) ,
sizeof(int)*DATA_BUFFER_DMA, tag_id, 0, 0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
  mfc_write_tag_mask(1<<tag_id);

  /* Wait for the data array DMA to complete. */
  mfc_read_tag_status_all();

  z=0;
/*Transferimos de la fracción de datos que hemos leido del SPE2 al
array principal más lo que hubiera antes ya*/
  for(j=i*DATA_BUFFER_DMA;j<(DATA_BUFFER_DMA+DATA_BUFFER_DMA*i);j++)
  {
        data_out[j] = data_out[j] + data_part[z];
        z++;
  }

}

//*****************************************************************
*****
for(i=0;i<234;i++)
{
/*Transferimos de la fracción de datos que vamos a transmitir al
proceso principal*/
  z=0;
  for(j=i*DATA_BUFFER_DMA;j<(DATA_BUFFER_DMA+DATA_BUFFER_DMA*i);j++)
  {
        data_part[z] = data_out[j];
        z++;
  }

/* DMA the data to system memory from our local store buffer. */

mfc_put(data_part,cb1.addr_out+(sizeof(int)*DATA_BUFFER_DMA*i),DATA_BU
FFER_DMA*sizeof(int),tag_id,0,0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
*/
  mfc_write_tag_mask(1<<tag_id);
```

```
  /* Now, issue the read and wait to guarantee DMA completion before
we  */
  /* continue. */
  mfc_read_tag_status_all();
}

  return 0;
}
```

- SPE2

```
#include <spu_mfcio.h>
#include <stdio.h>
#include <simple_dma.h>

#include <sched.h>
#include <pthread.h>
#include <dirent.h>
#include <stdint.h>
#include <errno.h>
#include <sys/wait.h>
#include <string.h>

/* Here's the local copy of the control block, to be filled by the DMA
*/
volatile control_block cb1 __attribute__ ((aligned (ALIGMENT)));

/* Here's the local copy of the data array, to be filled by the DMA */
int data_out[DATA_BUFFER_ENTRIES] __attribute__ ((aligned
(ALIGMENT)));
int data_part[DATA_BUFFER_DMA] __attribute__ ((aligned (ALIGMENT)));

int main(unsigned long long speid __attribute__ ((unused)),
       unsigned long long argp,
       unsigned long long envp __attribute__ ((unused)))
{

  int i,rc,j,z;
  unsigned int tag_id;
  unsigned long long addr;

  /* Reserve a tag for application usage */
  if ((tag_id = mfc_tag_reserve()) == MFC_TAG_INVALID) {
    printf("ERROR: unable to reserve a tag\n");
    return 1;
  }

  /* Here is the actual DMA call */
  /* the first parameter is the address in local store to place the
data */
  /* the second parameter holds the main memory address
*/
  /* the third parameter holds the number of bytes to DMA
*/
  /* the fourth parameter identifies a "tag" to associate with this
DMA  */
  /* (this should be a number between 0 and 31, inclusive)
*/
  /* the last two parameters are only useful if you've implemented
your  */
```

```
  /* own cache replacement management policy.  Otherwise set them to
0.  */
//  mfc_get(&cb1, argp, 128*sizeof(cb1), tag_id, 0, 0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
*/
  mfc_write_tag_mask(1<<tag_id);

  /* Now, issue the read and wait to guarantee DMA completion before
we  */
  /* continue. */
  mfc_read_tag_status_all();

  cb1.addr_in = 0x10050000;
  printf("Address received through control block2in = 0x%llx\n",
cb1.addr_in);

for(i=0;i<234;i++)
{
  /* DMA the data from system memory to our local store buffer. */
  mfc_get(data_part, cb1.addr_in +(DATA_BUFFER_DMA*i) ,
sizeof(int)*DATA_BUFFER_DMA, tag_id, 0, 0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
  mfc_write_tag_mask(1<<tag_id);

  /* Wait for the data array DMA to complete. */
  mfc_read_tag_status_all();

  z=0;
/*Transferimos de la fracción de datos que hemos leido al array
principal*/
  for(j=i*DATA_BUFFER_DMA;j<(DATA_BUFFER_DMA+DATA_BUFFER_DMA*i);j++)
  {
        data_out[j] = data_part[z];
        //data_part[z] = 0;
        z++;
  }

}


  cb1.addr_out = 0x10050000;
  printf("Address received through control block2out = 0x%llx\n",
cb1.addr_out);

//*****************************************************************
*****
for(i=0;i<234;i++)
{
z=0;
/*Transferimos de la fracción de datos que vamos a transmitir al
proceso principal*/
```

```
for(j=i*DATA_BUFFER_DMA;j<(DATA_BUFFER_DMA+DATA_BUFFER_DMA*i);j++)
{
      data_part[z] = data_out[j];
      z++;
}

/* DMA the data to system memory from our local store buffer. */

mfc_put(data_part,cb1.addr_out+(sizeof(int)*DATA_BUFFER_DMA*i),DATA_BU
FFER_DMA*sizeof(int),tag_id,0,0);

  /* Now, we set the "tag bit" into the correct channel on the
hardware  */
  /* this is always 1 left-shifted by the tag specified with the DMA
*/
  /* for whose completion you wish to wait.
*/
  mfc_write_tag_mask(1<<tag_id);

  /* Now, issue the read and wait to guarantee DMA completion before
we  */
  /* continue. */
  mfc_read_tag_status_all();

}
  return 0;
}
```

- Simple_dma: Include on estan definides totes les constants

```
#ifndef __simple_DMA_h__
#define __simple_DMA_h__

#include <stdlib.h>

#define DATA_BUFFER_ENTRIES     30000
#define DATA_BUFFER_SIZE      DATA_BUFFER_ENTRIES
#define DATA_BUFFER_DMA       128
#define ALIGMENT        65536

typedef struct _control_block {
  unsigned int  chunk_size; /* size, in bytes, of each of these array
pieces */
  unsigned char pad[4];     /* pad maintain alignment and sized for
correct DMA */
  unsigned long long addr_in;  /* address to be filled by single-
buffered DMA */
  unsigned long long addr_out;  /* address to be filled by single-
buffered DMA */
} control_block;
#endif /* __simple_DMA_h__ */
```

Bibliografia

(s.f.). Obtenido de https://www.ibm.com/developerworks/forums

(s.f.). Obtenido de http://www.idg.es/computerworld/articulo.asp?id=163469

(s.f.). Obtenido de http://www-01.ibm.com/support/docview.wss?rs=2021&context=SSVKBV&q1=mass_cbeppu_docs&uid=swg27009546&loc=en_US&cs=utf-8&lang=en

(s.f.). Obtenido de http://www.redbooks.ibm.com/abstracts/sg247575.html

(s.f.). Obtenido de http://www.guia-ubuntu.org/index.php?title=Listado_de_%C3%B3rdenes

(s.f.). Obtenido de http://www.ibm.com/developerworks/power/library/pa-linuxps3-1/

(s.f.). Obtenido de http://www.power.org/resources/devcorner/cellcorner/cellworkshop0606

(s.f.). Obtenido de http://www.elotrolado.net/hilo_ibm-cell-35-veces-mas-rapido-en-3d-que-un-g5_552971_s120

(s.f.). Obtenido de http://www.renderstate.de/?p=6

(s.f.). Obtenido de http://www-01.ibm.com/support/docview.wss?rs=2021&context=SSVKBV&dc=DA400&uid=swg27009548&loc=en_US&cs=UTF-8&lang=en&rss=ct2021other

(s.f.). Obtenido de http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingTutorial/

(s.f.). Obtenido de http://www.ibm.com/developerworks/library/pa-libspe2/

Arevalo, A., Matinata, R., Pandian, M., Peri, E., Ruby, K., Thomas, F., y otros. (2008). *Programming the Cell Broadband.*

IBM Corporation. (2007). *An introduction to compiling for the Cell Broadband.*

IBM Corporation. (2010). *Cell BE Software Development Kit 2.0.*

IBM Corporation. (2006). *Cell Broadband Engine Architecture Overview.*

IBM Corporation. (2008). *Cell Broadband Engine Programming Handbook.*

IBM Corporation. (2008). *Cell Broadband EngineTM .*

IBM Corporation. (2006). *Cell Software Development Environment.*

IBM Corporation. (2007). *Cell Software Model.*

IBM Corporation. (2007). Hands-on System Check.

IBM Corporation. (2006). *Hello World!*

IBM Corporation. (2008). *Installation Guide v3.1.*

IBM Corporation. (2006). *Software Development Toolkit Installation and User's GuideVersion 1.1.*

IBM Corporation. (2008). *SPE Runtime Management Library Version 2.3.*

IBM Corporation. (2005). *SPU C/C++ Language Extensions.*

IBM. *Programming Tutorial DRAFT.*

Innovative Computing Laboratory. (2008). *Sony/Toshiba/IBM (STI) CELL Processor.*

Kreibe, M., Longley, M., & Snyder, P. *CBE Architecture Overview.*

Oron, H. (2008). *Programació de la PS3 amb Linux.*

Sony Corporation. (2006). *SIMD Math Library Specification for Cell Broadband Engine™ Architecture.*

Stürmer, M. (2008). *PlayStation Programming 2008:.*

Universitat de València Tecnología de Computadoras. (2009). Práctica 4 Estudio del procesador Cell.