Le processeur CELL

Architecture et programmation

Jean-Luc.Lamotte@lip6.fr

Université P. et M. Curie
Laboratoire LIP6 - CNRS 7606

Organisation de l'exposé

- Architecture du processeur
- Programmation
  - échange de données
  - programmation SIMD

Historique

- **IBM, SCEI/Sony, Toshiba Alliance formed in 2000**
- **Design Center opened in March 2001**
  - Based in Austin, Texas
- **Single CellBE operational Spring 2004**
- **2-way SMP operational Summer 2004**
- **February 7, 2005: First technical disclosures**
- **October 6, 2005: Mercury announces Cell Blade**
- **November 9, 2005: Open source SDK & simulator published**
- **November 14, 2005: Mercury announces Turismo Cell offering**
- **February 8, 2006: IBM announced Cell Blade**
- **July 17, 2006: SDK 1.1 available**
- **October 17, 2007: SDK 3.0 available**

Les objectifs

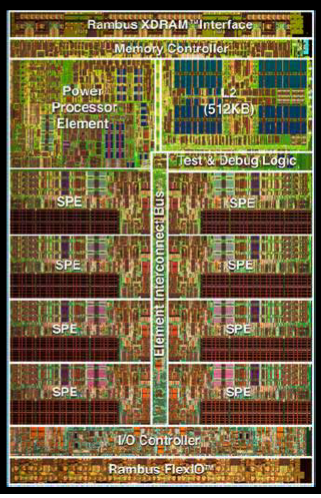
- **Compatible avec 64b Power Architecture™**
  - Réutilisation de tous les savoirs sur ce type d'architecture
- **Amélioration des performances et de l'efficacité en**
  - Attaquant le mur de la puissance
    - Utilisation de processeurs non homogène
    - Utilisation d'une fréquence élevée et d'une alimentation électrique à faible volatge
  - Attaquant le mur de l'accés à la mémoire
    - Streaming DMA architecture
    - Gestion mémoire à trois niveau : mémoire principale, mémoire local (LS), Register Files
  - Attaquant le mur de la fréquence
    - Highly optimized implementation
    - Large shared register files and software controlled branching to allow deeper pipelines
- **Multi-OS support, including RTOS / non-RTOS**

Le HPC et le CELL

- Convergence HPC/multimédia
- Roadrunner : première machine à atteindre 1 Pflops
- HPC : 100kcore -> 1Mcore
- Eflops, remise en cause des paradigmes de programmation
- Nouvelles problématiques à prendre en compte : les pannes
- Grosse équipe de développement sur de très gros codes
- Benchmark (vitesse, résultats)
- Longue durée de vie des codes

Highlights (3.2 GHz)

- 241M transistors
- 235mm2
- 9 cores, 10 threads
- >200 GFlops (SP)
- >20 GFlops (DP)
- Up to 25 GB/s memory B/W
- Up to 75 GB/s I/O B/W
- >300 GB/s EIB
- Top frequency >4GHz (observed in lab)

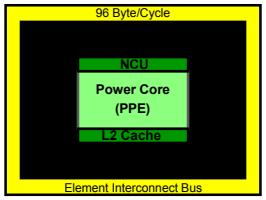


Cell Processor Components

Power Processor Element (PPE):

- General purpose, 64-bit RISC processor (PowerPC AS 2.0.2)
- 2-Way hardware multithreaded
- L1 : 32KB I ; 32KB D
- L2 : 512KB
- Coherent load / store
- VMX-32
- Realtime Controls
  - Locking L2 Cache & TLB
  - Software / hardware managed TLB
  - Bandwidth / Resource Reservation
  - Mediated Interrupts

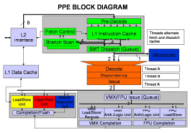
In the Beginning
– the solitary Power Processor

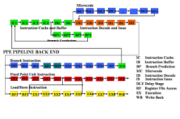


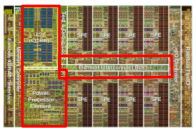
Custom Designed
– for high frequency, space, and power efficiency

Element Interconnect Bus (EIB):

- Four 16 byte data rings supporting multiple simultaneous transfers per ring
- 96Bytes/cycle peak bandwidth
- Over 100 outstanding requests



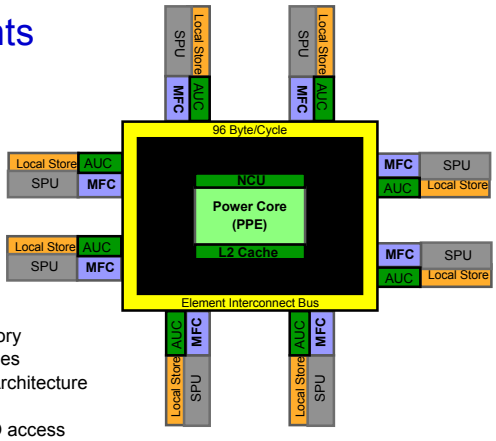




Cell Processor Components

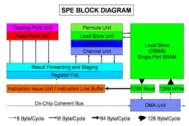
Synergistic Processor Element (SPE):

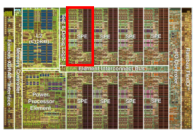
- Provides the computational performance
- Simple RISC User Mode Architecture
  - Dual issue VMX-like
  - Graphics SP-Float
  - IEEE DP-Float
- Dedicated resources: unified 128x128-bit RF, 256KB Local Store
- Dedicated DMA engine: Up to 16 outstanding requests



Memory Management & Mapping

- SPE Local Store aliased into PPE system memory
- MFC/MMU controls / protects SPE DMA accesses
  - Compatible with PowerPC Virtual Memory Architecture
  - SW controllable using PPE MMIO
- DMA 1,2,4,8,16,128 -> 16Kbyte transfers for I/O access
- Two

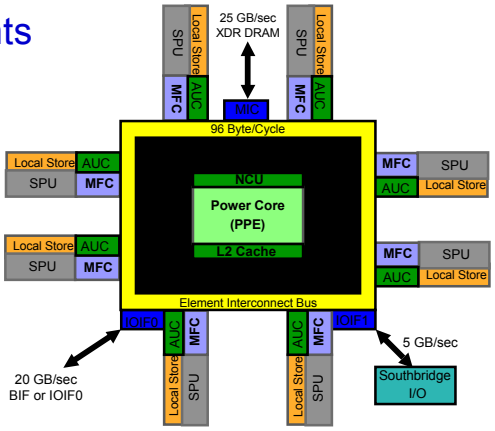




Cell Processor Components

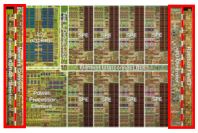
Broadband Interface Controller (BIC):

- Provides a wide connection to external devices
- Two configurable interfaces (60GB/s @ 5Gbps)
  - Configurable number of bytes
  - Coherent (BIF) and / or I/O (IOIFx) protocols
- Supports two virtual channels per interface
- Supports multiple system configurations

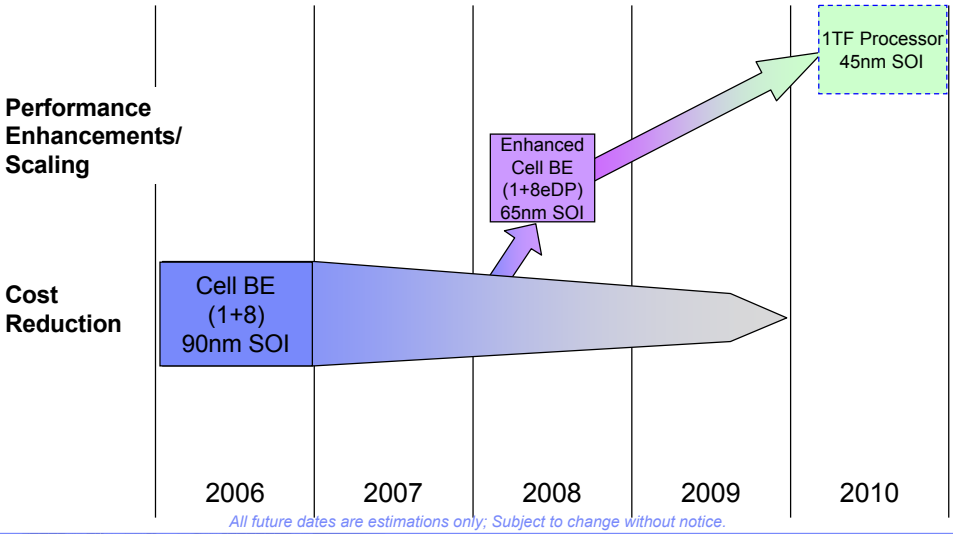


Memory Interface Controller (MIC):

- Dual XDR™ controller (25.6GB/s @ 3.2Gbps)
- ECC support
- Suspend to DRAM support



Cell Broadband Engine Architecture™ Technology Competitive Roadmap



All future dates are estimations only; Subject to change without notice.

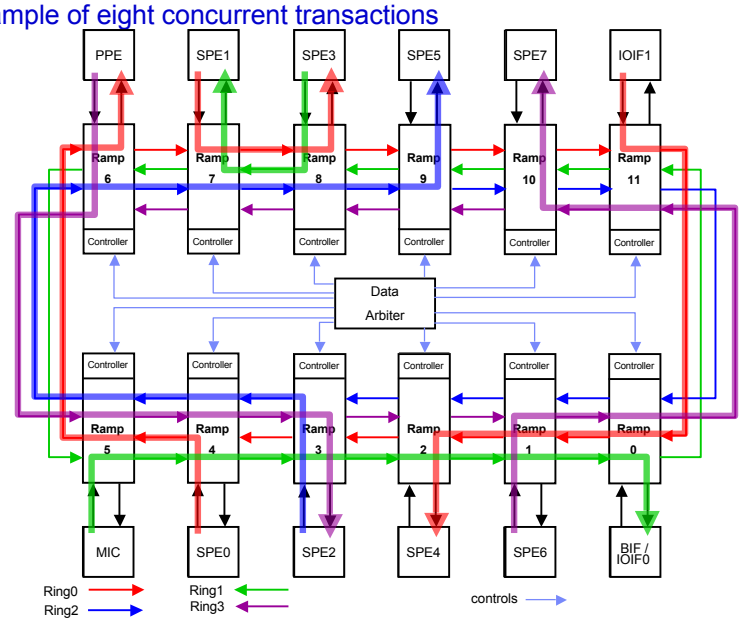
Internal Bandwidth Capability

- **Each EIB Bus data port supports 25.6GBytes/sec* *in each direction***
- **The EIB Command Bus streams commands fast enough to support 102.4 GB/sec for coherent commands, and 204.8 GB/sec for non-coherent commands.**
- **The EIB data rings can sustain 204.8GB/sec for certain workloads, with transient rates as high as 307.2GB/sec between bus units**

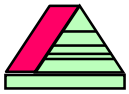
Despite all that available bandwidth…

* The above numbers assume a 3.2GHz core frequency – internal bandwidth scales with core frequency

Example of eight concurrent transactions

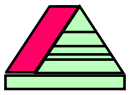


Code Development Tools



- **GNU based binutils**
  - From Sony Computer Entertainment
  - gas SPE assembler
  - gld SPE ELF object linker
    - ppu-embedspu script for embedding SPE object modules in PPE executables
  - misc bin utils (ar, nm, ...) targeting SPE modules
- **GNU based C/C++ compiler targeting SPE**
  - From Sony Computer Entertainment
  - retargeted compiler to SPE
  - Supports common SPE Language Extensions and ABI (ELF/Dwarf2)
- **Cell Broadband Engine Optimizing Compiler (executable)**
  - IBM XLC C/C++ for PowerPC (Tobey)
  - IBM XLC C retargeted to SPE assembler (including vector intrinsics) - highly optimizing
  - Prototype CBE Programmer Productivity Aids
    - Auto-Vectorization (auto-SIMD) for SPE and PPE Multimedia Extension code
  - spu_timing Timing Analysis Tool

SPE Performance Tools (executables)



- **Static analysis (spu_timing)**
  - Annotates assembly source with instruction pipeline state
- **Dynamic analysis (CBE System Simulator)**
  - Generates statistical data on SPE execution
    - Cycles, instructions, and CPI
    - Single/Dual issue rates
    - Stall statistics
    - Register usage
    - Instruction histogram

CELL Software Design Considerations

- **Two Levels of Parallelism**
  - Regular vector data that is SIMD-able
  - Independent tasks that may be executed in parallel
- **Computational**
  - SIMD engines on 8 SPEs and 1 PPE
  - Parallel sequence to be distributed over 8 SPE / 1 PPE
  - 256KB local store per SPE usage (data + code)
- **Communicational**
  - DMA and Bus bandwidth
    - DMA granularity – 128 bytes
    - DMA bandwidth among LS and System memory
  - Traffic control
    - Exploit computational complexity and data locality to lower data traffic requirement
  - Shared memory / Message passing abstraction overhead
  - Synchronization
  - DMA latency handling

Programmation du processeur CELL BE

Principes généraux

- Notion de contexte d'exécution
- Un *context* est exécuté sur 1 SPE jusqu'à sa fin. Il n'est pas possible d'exécuter en même temps plus de *context* qu'il y a de SPE
- Le *context* ne dispose que de 256 Ko (pas de mémoire partagée)

- Dans l'ordre, on aura:
  - Création d'un contexte d'exécution (`spe_context_create`)
  - Chargement d'un programme (`spe_load_program`)
  - Lancement du context (`spe_context_run`). La fonction attend la fin du programme pour rendre la main

Nécessité d'utiliser les pthread pour faire du parallélisme sur les SPE


Hello world (PPE, 1 SPE), coté PPE

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

/* Ce programme cree 1 context qui ecrit hello world sur la sortie
   standard */

#define PERROR(s) {perror(s); exit(1);}

/* Declaration du code qui tournera sur les SPE */
extern spe_program_handle_t hello1_spu;

/* On a au maxi 6 THREADS sur les PS3 */
/* sur les autres cartes, on peut ajour jusqu'a 2 Cells donc 16
   threads */

#define MAX_SPE   1
```


```
int main()
{
  uint32_t nbspe;
  spe_context_ptr_t context;
  uint32_t entry = SPE_DEFAULT_ENTRY;

  /* recuperation du nombre de SPE */
  nbspe = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
  if (nbspe < MAX_SPE) PERROR("pas suffisamment de SPE");

  /* CONTEXT creation  */
  if ((context = spe_context_create (0, NULL)) == NULL)
    PERROR ("creation context");

  if (spe_program_load (context, &hello1_spu))
    PERROR ("chargement du code du thread");

  if (spe_context_run(context, &entry, 0, NULL, NULL, NULL) < 0)
    PERROR ("Failed running context");

 /* CONTEXT Destruction  */
  if (spe_context_destroy (context) != 0)
    PERROR("probleme destroying context");

  printf("C'est fini\n");
  return (0);
}
```

Hello Word, coté SPE

```
#include <stdio.h>

/* un petit hello world sur les SPE */

int main(unsigned long long id)
{
  printf("Hello world, je suis 0x%llx \n", id);
  return 0;
}
```

```
Hello world, je suis 0x10018010
C'est fini
```

Hello World (1PPE, nSPE)

```
extern spe_program_handle_t hello2_spu, bonjour_spu;
int i, nbspe;
 spe_context_ptr_t context[MAX_SPE];
 spe_program_handle_t *spethread;
 pthread_t threads[MAX_SPE];

 /* recuperation du nombre de SPE */
 nbspe = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
 printf("il y a %d SPES\n",nbspe);

 /* Les threads pairs ecrivent bonjour, les impairs hello world */
 for(i=0; i<nbspe; i++) {
   if ((context[i] = spe_context_create (0, NULL)) == NULL)
     PERROR ("creation context");

   /* on choisit hello world ou bonjour */
   if (i%2) spethread=&hello_spu;
   else spethread=&bonjour_spu;

   if (spe_program_load (context[i], spethread))
     PERROR ("chargement du code du thread");
   if (pthread_create (&threads[i], NULL, &ppu_pthread_function,
  &context[i]))   PERROR ("creation thread");
 }
```


Hello World (1PPE, nSPE)

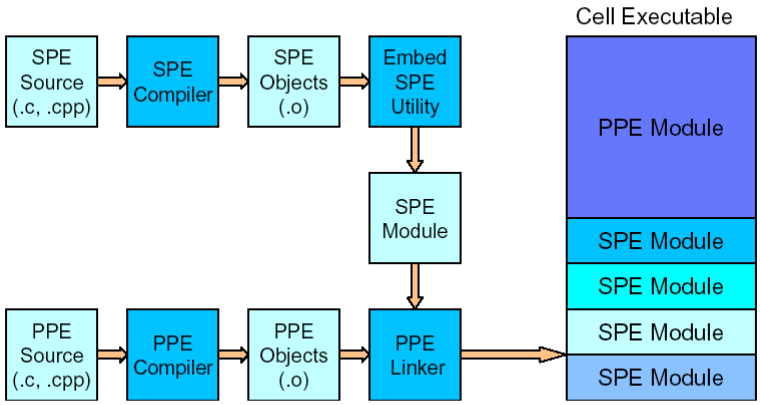
```
/* attente de la fin d'execution  */

 for (i=0; i<nbspe; i++) {
    /* on verifie la fin du thread */
    if (pthread_join (threads[i], NULL))
      PERROR("probleme pthread_join");

    /* Destruction du contexte */
    if (spe_context_destroy (context[i]) != 0)
      PERROR("probleme destroying context");
  }
  printf("C'est fini\n");
  return (0);
}
/* fonction generique extraite du tutorial IBM du SDK 3.0 */
void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  uint32_t 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");
  pthread_exit(NULL);
}
```


Build Process



✓Make scripts are available to automate the build process

Exécution du programme

```
il y a 6 SPES
Bonjour, je suis 0x1001e010
Hello world, I am 0x1001e460
Bonjour, je suis 0x1001e870
Hello world, I am 0x1001ec80
Bonjour, je suis 0x1001f090
Hello world, I am 0x1001f4a0
C'est fini
```

Les communications au sein du processeur CELL

- 3 possibilités :
  - Les *boîtes mail* sont utilisées pour le contrôle des communications entre les SPE, le PPE et les autres unités. Ce sont des messages de 32 bits. Chaque SPE dispose de 2 boîtes d'envoi et d'une boite de réception.
  - Les *transferts DMA* sont utilisés pour déplacer des instructions ou des données entre la mémoire et le LS. Les transferts sont synchrones ou asynchrones (afin de masquer la latence mémoire et le transfert par des calculs).
  - L'envoi de signaux est utilisé par le PPE ou d'autres périphériques.


La communication avec les *mailboxes*

Chaque thread possède 3 mailbox, deux en envoi et une en réception.
Les messages envoyés sont destinés au PPE et les messages reçus proviennent du PPE.
Le nom des fonctions est centré sur le SPE.
Quelques fonctions pour accéder à ces mailboxes.

- `(uint32_t) spu_read_in_mbox(void)` : la donnée suivante dans la « SPU InBound Mailbox » est lue. La commande échoue si la mailbox est vide.
- `(uint32_t) spu_stat_in_mbox(void)` : renvoie le nombre de message (de données) dans la « SPU InBound Mailbox ».
- `(void) spu_write_out_mbox(uint32_t data)` : la donnée est envoyée à « SPU OutBound Mailbox ». La commande échoue si la mailbox est vide.
- `(uint32_t) spu_stat_out_mbox(void)` : renvoie le nombre de message (de données) dans la « SPU OutBound Mailbox ».

- Deux autres fonctions avec interruption
  - `(void) spu_write_out_intr_mbox(uint32_t data)` : la donnée est envoyée à « SPU OutBound Interrupt Mailbox ». La commande échoue si la mailbox est vide.
  - `(uint32_t) spu_stat_out_intr_mbox(void)` : renvoie le nombre de message (de données) dans la « SPU OutBound Interrupt Mailbox ».


Mailbox côté PPE

- `int spe_in_mbox_write(spe_context_ptr_t c, unsigned int *data, int count, unsigned int behaviour)` : place la donnée `data` dans la InBounding Mailbox du thread spécifié. Si la boite est pleine, la dernière entrée est écrasée.
- `int spe_out_mbox_read(spe_context_ptr_t c, unsigned int *data, int count)` envoie le nombre de donnée lue dans de la SPU outbounding mailbox pour le contexte `c, -1 est retourné en cas d'erreur`
- `(int) spe_in_mbox_status(spe_context_ptr_t c)` : la fonction renvoie le statut de la SPU inbound mailbox pour le thread spécifié. 0 est retourné si la mailbox est pleine, >0 nombre de données 32 bits qu'il est possible d'écrire.
- `(int) spe_out_mbox_status(spe_context_ptr_t c)` : la fonction renvoie le statut de la SPU outbound mailbox pour le thread spécifié. 0 est retourné si la mailbox est vide. >0 nombre de données lisibles.

- Attention : il semble que 4 messages suffisent à remplir la mailbox. Il faut dans ce cas vérifier que la mailbox n'est pas pleine avant d'envoyer un mail.
- Même fonction avec les interruptions.


Aparté sur la gestion de la mémoire

- Postulat : 1 thread s 'execute sur 1 SPE jusqu'à sa fin.

- EA (effective address) correspond à une adresse dans la mémoire partagée. Elle est codée sur 64 bits.
- Les adresses côté PPE sont des adresses sur 64 bits.

- LS (local Storage) : vu du côté du SPE, les adresses dans le LS sont des adresses relatives codées sur 32 bits. Cependant la mémoire est mappée dans la mémoire centrale. Le PPE peut connaître l'adresse de début de la mémoire du SPE avec la fonction `spe_ls_area_get(spe_context_ptr_t)`
- Pour que le PPE envoie une donnée au SPE, il doit donc attendre de connaître l'adresse (sur 32 bits) du buffer de réception sur le SPE. Il calcule ensuite l'adresse EA de ce buffer en l'ajoutant à l'adresse de début du thread. Il peut alors écrire la donnée dans la bonne zone mémoire.


Manipulation des addresses par les programmes « SPE »

mfc_ea2h: Extract Higher 32 Bits from Effective Address

`(uint32_t) mfc_ea2h(uint64_t ea)`

The higher 32 bits are extracted from the 64-bit effective address *ea*.
Implementation `(uint32_t)((uint64_t)(ea)>>32)`

mfc_ea2l: Extract Lower 32 Bits from Effective Address

`(uint32_t) mfc_ea2l(uint64_t ea)`

The lower 32 bits are extracted from the 64-bit effective address *ea*.
Implementation `(uint32_t)(ea)` `

mfc_hl2ea: Concatenate Higher 32 Bits and Lower 32 Bits

`(uint64_t) mfc_hl2ea(uint32_t high, uint32_t low)`

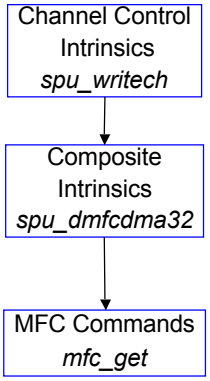
The higher 32 bits of a 64-bit address *high* and the lower 32 bits *low* are concatenated.

#include `<spu_mfcio.h>`

DMA Commands

- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**

DMA Commands



defined as macros in spu_mfcio.h

For details see: SPU C/C++ Language Extensions

DMA Characteristics

- **DMA transfers**
  - transfer sizes can be 1, 2, 4, 8, and n*16 bytes (n integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable
- **DMA command queues per SPU**
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
    - SPU-initiated DMA is always preferable
- **DMA tags**
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands
- **DMA lists**
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests

Transfer from PPE (Main Memory) to SPE

- **DMA get from main memory**

  mfc_get(lsaddr,ea,size,tag_id,tid,rid);
  - lsaddr = target address in SPU local store for fetched data (SPU local address)
  - ea = effective address from which data is fetched (global address)
  - size = transfer size in bytes
  - tag_id = tag-group identifier
  - tid = transfer-class id
  - rid = replacement-class id
- **Also available via "composite intrinsic":**

  spu_mfcdma64(lsaddr, eahi, ealow, size, tag_id, cmd);

DMA Command Status (SPE)

- **DMA read and write commands are non-blocking**
- **Tags, tag groups, and tag masks used for:**
  - checking status of DMA commands
  - waiting for completion of DMA commands
- **Each DMA command has a 5-bit tag**
  - commands with same tag value form a "tag group"
- **Tag mask is used to identify tag groups for status checks**
  - tag mask is a 32-bit word
  - each bit in the tag mask corresponds to a specific tag id:

    tag_mask = (1 << tag_id)

DMA Tag Status (SPE)

- **Set tag mask**

  unsigned int tag_mask;

  mfc_write_tag_mask(tag_mask);

  – tag mask remains set until changed
- **Fetch tag status**

  unsigned int result;

  result = mfc_stat_tag_status();

  – tag status Is logically ANDed with current tag mask

  – tag status bit of '1' indicates that no DMA requests tagged with the specific tag id (corresponding to the status bit location) are still either in progress or in the DMA queue

Waiting for DMA Completion (SPE)

- **Wait for any tagged DMA:**

  mfc_read_tag_status_any():

  – wait until **any** of the specified tagged DMA commands is completed
- **Wait for all tagged DMA:**

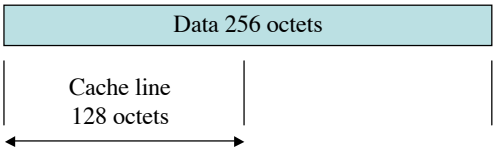
  mfc_read_tag_status_all():

  – wait until **all** of the specified tagged DMA commands are completed

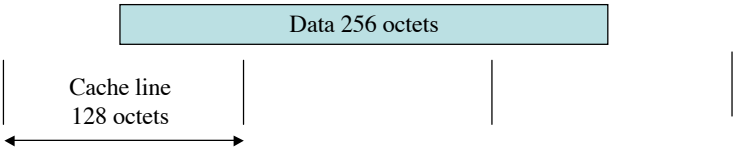
➢ **Specified tagged DMA commands = commands specified by current tag mask setting**

Gestion des caches

Transfert efficace : 2 remplissages de cache pour un transfert



Transfert inefficace : 3 remplissages de cache sont nécessaires.
Les données ne sont pas alignées sur un cache line



Moralité : il faut toujours aligner vos données pour être efficace en utilisant: `malloc_align(taille, 7)`

Ou `_attribute_ (aligned(128))`

Un échange complet entre PPE et SPE

Le PPE lance ACTIVE threads. Il attend de recevoir de chaque thread l'adresse d'un buffer. Dans ce buffer, il écrit un texte et il envoie un mail a chaque thread pour lui dire que son buffer est rempli. Chaque thread écrit le texte reçu sur la sortie standard, le modifie et met a jour la mémoire principale. A la fin de l'éxecution, le PPE affiche le contenu des buffers

Programme SPE

```
int main(unsigned long long speid)
{
  uint32_t adrl, adrh;
  uint64_t adresse;
  char contenu[128] __attribute__((aligned (128)));
  int tag = 1, tag_mask = 1<<tag;
  //  reception des parties haute et basse d'une adresse memoire
  while (spu_stat_in_mbox()==0);
  adrh=spu_read_in_mbox();
  adrl=spu_read_in_mbox();

  // reconstruction de l'adresse
  adresse=mfc_hl2ea(adrh,adrl);

  printf("SPE %lld recu l'adrh %x adrl %x\n",speid,adrh, adrl);
  printf("SPE %lld recu l'adresse %llx\n",speid,adresse);

  // acces DMA pour recuperer une zone memoire en local sur le SPE
  mfc_get(contenu, adresse, 128, tag, 0, 0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_any();

  printf("SPE %lld contenu=%s\n",speid, contenu);
  strcat(contenu,", Bonjour PPE");

  // acces DMA pour ecrire en memoire le contenu d'une variable d'un SPE
  mfc_put(contenu, adresse, 128, tag, 0, 0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_any();
  return 0;
```

Le programme du PPE

```
int main()
{
  uint32_t i, nbspe, tmp;
  spe_context_ptr_t context[ACTIVE];
  pthread_t threads[ACTIVE];
  char *message[ACTIVE];

  /* recuperation du nombre de SPE */

  /* le lancement des threads a été supprimé du listing
```

```
 /* La partie du code qui interagit avec les threads */
  for (i=0;i<ACTIVE;i++){
    // allocation d'un tableau de taille 128 (un multiple de 16)
    // sur une adresse multiple de 128 = pow(2,7)
    if ((message[i]=_malloc_align(128,7))==NULL)
         PERROR("Allocation memoire");
    sprintf(message[i],"bonjour thread %d ",i);
    printf("PPE adresse message[%d]=%llx\n",i, (unsigned long long )
   message[i])
;
    // On coupe l'adresse de 64 bits en deux parties
    // que l'on envoie avec un "mail"
    tmp=((uint64_t)message[i]>>32);
    spe_in_mbox_write(context[i], &tmp, 1,SPE_MBOX_ALL_BLOCKING);
    // pour eviter un warning
    tmp=(uint32_t) (uint64_t) (message[i]);
    spe_in_mbox_write(context[i], &tmp,1,SPE_MBOX_ALL_BLOCKING);
  }
  // synchronisation de la fin des threads supprimées
  …

for (i=0;i<ACTIVE;i++)
    printf("PPE : message[%d]=%s\n",i,message[i]);

printf("C'est fini\n");
```


Trace exécution

```
PPE adresse message[0]=10019880
SPE 268537872 recu l'adrh 0 adrl 10019880
SPE 268537872 recu l'adresse 10019880
SPE 268537872 contenu=bonjour thread 0

PPE adresse message[1]=10019a00
SPE 268538912 recu l'adrh 0 adrl 10019a00
SPE 268538912 recu l'adresse 10019a00
SPE 268538912 contenu=bonjour thread 1

PPE fin des threads
PPE : message[0]=bonjour thread 0 , Bonjour PPE
PPE : message[1]=bonjour thread 1 , Bonjour PPE
C'est fini
```


Utilisation des arguments de la fonction `main`

Prototype de la fonction main d'un programme s'exécutant sur un SPE:

```
int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp)
```

Prototype de la fonction `spe_context_run`

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

Utilisation d'un « control block » qui permet de passer différentes valeurs aux SPE sans utiliser les envois d'adresses par les mailbox.

Le « control block » doit avoir une taille multiple de 16 octets et être inférieure à 16Ko

Il est intéressant d'aligner le CB


Exemple d'utilisation du CB

Le PPE crée un buffer qui contient un texte commun et ACTIVE buffers dans lequel il copie un texte particulier. Les pointeurs sur ces buffers sont stockés dans une structure de données control block (CB). Lors de la création des threads, un pointeur sur le CB est passé en argument à chaque thread. Chaque thread fait un accés DMA pour récuperer le contenu du CB et d'autres accés DMA pour récuperer le contenu des buffers.

Le fichier arg.h

```
struct t_arg{
  spe_context_ptr_t context;
  struct t_cb *cblock;
};
```

Le fichier cb.h

```
struct t_cb{
  // attention, la taille doit
  // etre multiple de 16
  unsigned int numproc;
  unsigned int remplissage;
  unsigned long long local;
  unsigned long long commun;
  unsigned long long remplissage1;

};

#define TAILLEMESSAGE 256
```


Partie PPU

```
int main()
{
  uint32_t i, nbspe;
  pthread_t threads[ACTIVE];
  char commun[TAILLEMESSAGE] __attribute__ ((aligned (128)));
  struct t_arg arg[ACTIVE];

  strcpy(commun,"PARTIE COMMUNE");

  /* recuperation du nombre de SPE */
  nbspe = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
  if (nbspe < ACTIVE) PERROR("pas assez de SPE");

  // remplissage du control block
  for(i=0; i<ACTIVE; i++) {
    // ALIGNEMENT FONDAMENTAL
    arg[i].cblock=(struct t_cb *)_malloc_align(sizeof(struct t_cb),7);
    (arg[i].cblock)->local=(unsigned long long)_malloc_align(TAILLEMESSAGE,7);
    sprintf((char*)(arg[i].cblock)->local,"partie locale du thread %d",i);
    (arg[i].cblock)->commun=(unsigned long long)commun;
    (arg[i].cblock)->numproc=i;
    printf("PPE %d %s %s\n",(arg[i].cblock)->numproc,
           (char *)(arg[i].cblock)->commun, (char*)(arg[i].cblock)->local);
  }
```


```
for(i=0; i<ACTIVE; i++) {
   printf("PPE : creation du contexte %u\n",(arg[i].cblock)->numproc);
   if ((arg[i].context = spe_context_create (0, NULL)) == NULL)
     PERROR ("creation context");

   printf("PPE 1 adresse cb=%llx\n",(unsigned long long)arg[i].cblock);
   printf("PPE 1 adresse context=%llx\n",(unsigned long long)arg[i].context);

   printf("PPE chargement du programme\n");
   if (spe_program_load (arg[i].context, &echangeCB_spu))
     PERROR ("chargement du code du thread");

   printf("PPE creation du thread\n");
   if (pthread_create (&threads[i], NULL, &ppu_pthread_function, &arg[i]))
     PERROR ("creation thread");
 }

 printf("PPE attente fin des threads\n");

 /* attente de la fin d'execution  */
 for (i=0; i<ACTIVE; i++) {
   /* on verifie la fin du thread */
   if (pthread_join (threads[i], NULL))
     PERROR("probleme pthread_join");

   /* Destruction du contexte */
   if (spe_context_destroy (arg[i].context) != 0)
     PERROR("probleme destroying context");
 }
 printf("C'est fini\n");
 return (0);
```


Coté SPU

```
#include "../cb.h"

int main(uint64_t speid,
         uint64_t argp)
{
  struct t_cb cb ;//__attribute__ ((aligned (128)));

  char mescom[TAILLEMESSAGE] __attribute__ ((aligned (128)));
  char mesloc[TAILLEMESSAGE] __attribute__ ((aligned (128)));

  uint32_t tag = 1, tag_mask = 1<<tag;

  // on va chercher le cb en memoire

  printf("SPE %llx recu du CB arg=%llx\n",speid,argp);

  mfc_get(&cb, argp, sizeof(cb), tag, 0, 0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_all();

  mfc_get(mescom, cb.commun, TAILLEMESSAGE, tag, 0, 0);
  mfc_get(mesloc, cb.local,  TAILLEMESSAGE, tag, 0, 0);
  mfc_read_tag_status_all();

  printf("SPEID %llu, numproc %u\n",speid, cb.numproc);
  printf("SPEID %u,  commun %s local %s\n",cb.numproc, mescom, mesloc);

  return 0;
}
```

Trace d'exécution

- PPE 0 PARTIE COMMUNE partie locale du thread 0
- PPE : creation du contexte 0
- PPE 1 adresse cb=10018080
- PPE 1 adresse context=10018270
- PPE chargement du programme
- PPE creation du thread
- PPE 2 adresse cb=10018080
- PPE 2 adresse context=10018270
- SPE 10018270 recu du CB arg=10018080
- PPE attente fin des threads
- SPEID 268534384, numproc 0
- SPEID 0, commun PARTIE COMMUNE local partie locale du thread 0
- C'est fini

Mais encore

- Les listes pour les accés DMA
- Les accés atomiques à la mémoire
- Les mécanismes de cohérence de mémoire entre PPE et SPE
- Les techniques de double buffering pour limiter le surcoût des communications
- Les environnements de plus haut niveau : DaCS, ALF

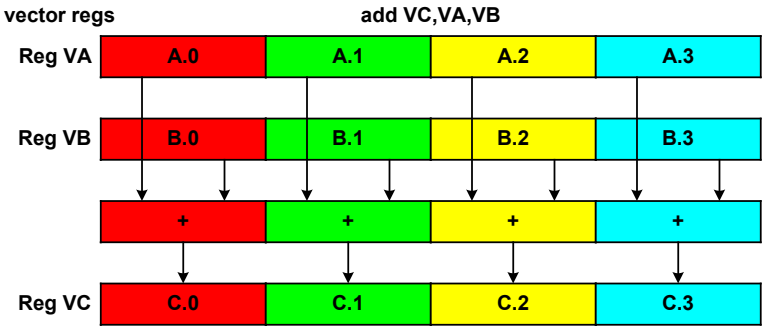
La programmation du processeur SIMD du SPE

- Calcul uniquement sur les vecteurs
- 128 registres de 128 bits
- Double pipelines
- Grand importance à avoir des instructions indépendants afin de les enchainer dans le pipeline.
- FMA *(ax+b)*

A lire : C/C++ Language Extensions for Cell Broadband Engine Architecture

http://www01.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E/$file/Language_Extensions_for_CBEA_2.4.pdf

Les registres vectoriels



Exemple d'opérations entre deux registres vectoriels sur des nombres 32 bits.

Les 4 opérations sont effectuées en même temps

Les types vecteurs

Type	Dimension
`vector unsigned char`	16 valeurs 8 bits
`vector int`	4 entiers de 32 bits
`vector short`	8 entiers de 16 bits
`vector float`	4 flottants de 32 bits
`vector double`	2 flottants 64 bits

Il existe d'autres types.

Attention, tous les types ne sont pas compatibles entre le SPU et le PPU.

Avec `vector X* p= &a; p+1` pointe sur le bloc vecteur suivant (16 octets).

Les *Intrinsics*

Il existe 3 classes d'extension intrinsics en C/C++.

- *Specific Intrinsics*
  - Il y une correspondance directe entre une instruction assembleur
  - Description plus lourde
  - Prefixe si_... Exemple si_stop
  - Intrinsics that have a one-to-one mapping with a single assembly-language instruction. Programmers rarely need the specific intrinsics for implementing inline assembly code because most instructions are accessible through generic intrinsics.
  - Prefixe si_ , e.g. si_stop
- *Generic Intrinsics et Built-Ins*
  - Plusieurs instructions assembleurs sont utilisées pour faire une instructions de plus haut niveau.
  - C'est le niveau que nous utiliserons.
  - Prefixe spu_ , e.g. spu_add
- *Composite Intrinsics*
  - Niveau au dessus du précédent.

Quelques fonctions intrinsics de calcul

a,b,c,d : vecteur de float ou de double

d=spu_add(a,b)	a+b
d=sup_madd(a,b,c)	a*b+c
d=sup_msub(a,b,c)	a*b-c
d=spu_mul(a,b)	a*b
d=sup_nmadd(a,b,c)	-(a*b+c)
d=sup_nmsub(a,b,c)	-(a*b-c)
d=spu_sub(a,b)	a-b
d=spu_re(a)	Esti. Réciproque (1/x) (12 bits)
d=spu_rsqrte(a)	Esti. Recip. Sqrt (12 bits)

Certaines de ces fonctions sont utilisables sur d'autres types (voir la documentation)

Il n'y a pas de division. Il faut l'implémenter soi-même.

Exemple d'utilisation : la multiplication de vecteur

```
void mul(float *a, float *b,
         float *r, int n){
  int i;



  for(i=0;i<n;i++)
    r[i]=a[i]*b[i];
}
```

```
void vmul(float *a, float *b, float *r,
   int n){
  int i;
   vector float va=(vector float *)a;
   vector float vb=(vector float *)b;
   vector float vr=(vector float *)r;
   n=n/4;
  for(i=0;i<n;i++)
    vr[i]=spu_mul(va[i],b[i]);
}
```

- N doit être divisible par 4, ou il faut traiter un cas particulier
- Les tableaux doivent être alignés en mémoire pour avoir le maximum de performance

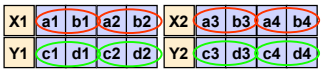

Autre exemple : multiplication complexe

Soit x et y, 2 complexes avec $x=a+ib$ et $y=c+id$

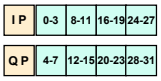
Rappel : $x*y=(a+ib)(c+id)=ac-bd+i(ad+bc)$

```
void cmult(float x[], float y[],float res, int n)
{
  int i;
  for(i=0;i<n;i++) {
    res[i*2]=x[i*2]*y[i*2] - x[i*2+1]*y[i*2+1];
    res[i*2+1]= x[i*2]*y[i*2+1] + x[i*2+1]*y[i*2];
}
```

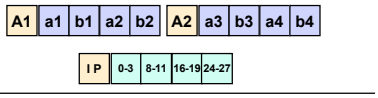

Programmation sur le SPE



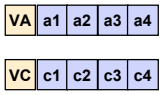
Shuffle patterns



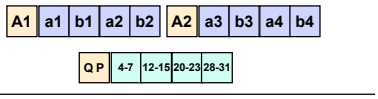

```
VA = spu_shuffle(X1, X2, IP);
```



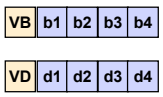

```
VC = spu_shuffle(Y1, Y2, IP);
```



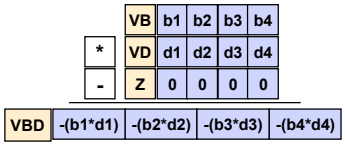

```
VB = spu_shuffle(X1, X2, QP);
```



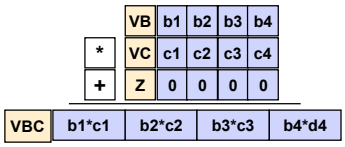

```
VD = spu_shuffle(Y1, Y2, QP);
```



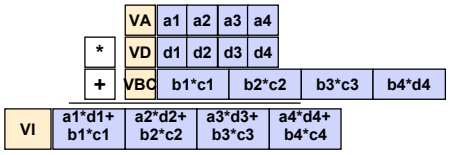

```
VBD = spu_nmsub(VB, VD, v_zero);
```



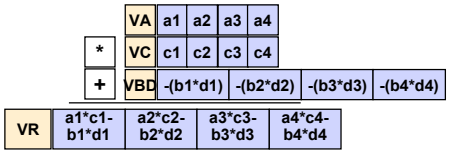

```
VBC = spu_madd(VB,VC, v_zero);
```

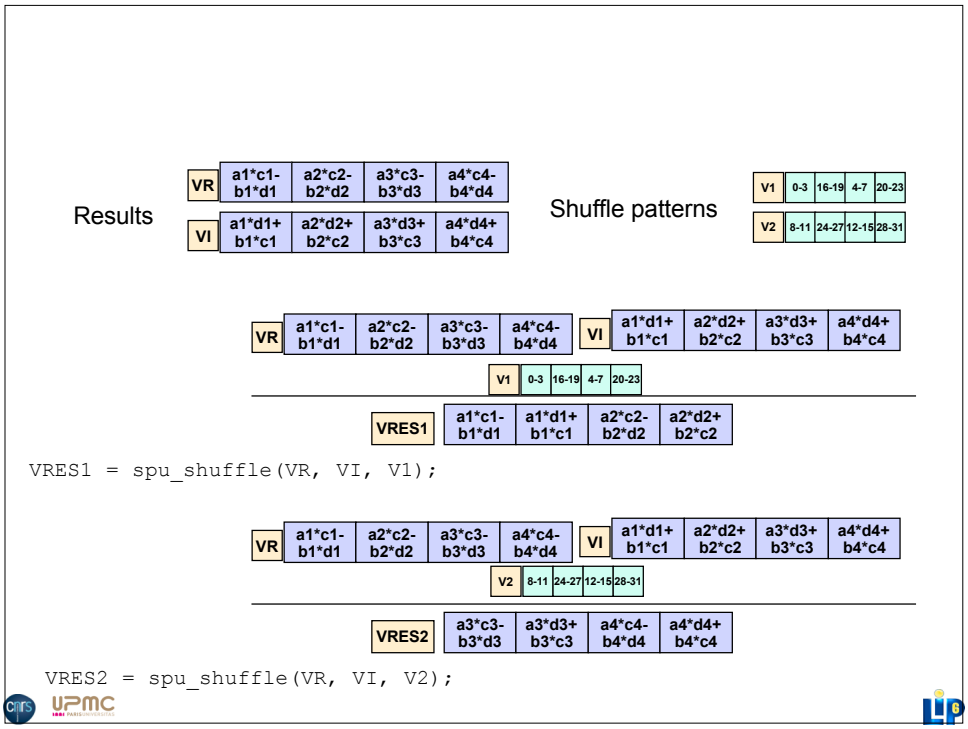



```
VI = spu_madd(VA, VD, VBC);
```




```
VR = spu_madd(VA, VC, VBD);
```






```
VRES1 = spu_shuffle(VR, VI, V1);
```

```
VRES2 = spu_shuffle(VR, VI, V2);
```

```
vector float X1, X2, Y1, Y2, VA, VB, VC, VD, VRES1, VRES2;
vector float v_zero = (vector float)(0,0,0,0);
vector unsigned char IP = (vector unsigned char)
              (0,1,2,3,8,9,10,11,16,17,18,19,24,25,26,27);
vector unsigned char QP = (vector unsigned char)
              (4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31);
vector unsigned char RP = (vector unsigned char)
              (0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23);
vector unsigned char SP = (vector unsigned char)
              (8,9,10,11,24,25,26,27,12,13,14,15,28,29,30,31);
VA = spu_shuffle(X1, X2, IP);
VB = spu_shuffle(Y1, Y2, IP);
VC = spu_shuffle(X1, X2, QP);
VD = spu_shuffle(Y1, Y2, QP);
VBD = spu_nmsub(VB, VD, v_zero);
VBC = spu_madd(VB, VC, v_zero);
VI = spu_madd(VA, VD, VBC);
VR = spu_madd(VA, VC, VBD);
VRES1 = spu_shuffle(VR, VI, RP);
VRES2 = spu_shuffle(VR, VI, SP);
```

Nb cycles des instructions

Instructions	Pipeline	latence
arithmétique, logique, comparaison, sélection	pair	2
opérations flottantes en simple précision (FMA) sauf division	pair	6
FMA sur des entiers de 16-bits	pair	7
Opérations flottantes en double précision	pair	6+7
décalage/rotation, estimation	impair	4
charge, stockage, canal	impair	6
débranchement	impair	1-18

Calcul du nombre de cycle

```
VA = spu_shuffle(A1, A2, IP);        0123
VB = spu_shuffle(B1, B2, IP);         1234
VC = spu_shuffle(A1, A2, QP);          2345          - cycle perdu
VD = spu_shuffle(B1, B2, QP);           3456
VBD = spu_nmsub(VB, VD, v_zero);         ---789012
VBC = spu_madd(VB, VC, v_zero);              890123
VI = spu_madd(VA, VD, VBC);                   -----456789
VR = spu_madd(VA, VC, VBD);                         567890
VRES1 = spu_shuffle(VR, VI, vcvmrgh);                -----1234
VRES2 = spu_shuffle(VR, VI, vcvmrgl);01                     2345
```

26 cycles.

Les valeurs sont déjà chargés dans les File Regiters

Calcul du nombre de cycle, nouvelle écriture

```
VB = spu_shuffle(B1, B2, IP);        0123
VD = spu_shuffle(B1, B2, QP);         1234
VC = spu_shuffle(A1, A2, QP);          2345
VA = spu_shuffle(A1, A2, IP);           3456
VBD = spu_nmsub(VB, VD, v_zero);         -567890
VBC = spu_madd(VB, VC, v_zero);            678901
VR = spu_madd(VA, VC, VBD);                 ----123456
VI = spu_madd(VA, VD, VBC);                      234567
VRES1 = spu_shuffle(VR, VI, vcvmrgh);             -----8901
VRES2 = spu_shuffle(VR, VI, vcvmrgl);01                  9012
```

23 cycles soit 10% de gain de performance

Travailler sur plus de données pour boucher les bulles

Optimisation de code

- Le SPE contient deux pipelines d'instructions, chaque instruction est pré-assignée à un unique pipeline par le compilateur
- Deux instructions peuvent être lancées pour chaque cycle d'horloge à condition que :
  - aucune dépendance entre ces deux instructions
  - les opérandes sont disponibles
  - l'instruction à l'adresse paire (les 3 derniers bits d'adresse sont 000) est une instruction de pipeline 0
  - l'instruction à l'adresse impaire (les 3 derniers bits d'adresse sont 100) est une instruction de pipeline 1 et
- Les instructions sont affectée dans l'ordre au pipeline 0 suivi par pipeline 1.

Repenser le code pour optimiser les 2 pipelines :

entrelacer les échanges memoire/registres et

les opérations

Conclusion

- Une architecture très différente des GPU
- Obligation de gérer beaucoup de problème
  - le recouvrement entre les accès mémoire et les calculs à l'opposé des GPU
  - vectorisation
- Doit pouvoir traiter des problèmes plus irréguliers que les GPU
- Nécessité de se rapprocher de la programmation assembleur
- Un avenir incertain.

- http:://pequan.lip6.fr/~lamotte/cell

Conclusions HPC et perspectives

- Complexité du processeur
- Rapport accès à la mémoire / vitesse de calcul
- Grand nombre d'outils pour l'abstraction du matériel : ALF, Dacs, MPI, OpenMP, ….
- Importance de bonnes bibliothèques de calcul BLAS, LAPACK
- Tuning de code difficile

- Nouvelles architectures de calcul émergeantes : GPGPU Fermi, LARRABEE
- Nouveaux environnement : OpenCL, 2009

Web Sites

- **Cell resource center at developerWorks**
  - http://www-128.ibm.com/developerworks/power/cell/
- **Cell developer's corner at power.org**
  - http://www.power.org/resources/devcorner/cellcorner/
- **The cell project at IBM Research**
  - http://www.research.ibm.com/cell/
- **The Cell BE at IBM alphaWorks**
  - http://www.alphaworks.ibm.com/topics/cell
- **Cell BE at IBM Engineering & Technical Services**
  - http://www-03.ibm.com/technology/
- **IBM Power Architecture**
  - http://www-03.ibm.com/chips/power/
- **Cell BE documentation at IBM Microelectronics**
  - http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_EngineCell
- **Linux info at the Barcelona Supercomputing Center website**
  - http://www.bsc.es/projects/deepcomputing/linuxoncell/

Systems and Technology Group

Web Sites

- **Cell resource center at developerWorks**
  - http://www-128.ibm.com/developerworks/power/cell/
- **Cell developer's corner at power.org**
  - http://www.power.org/resources/devcorner/cellcorner/
- **The cell project at IBM Research**
  - http://www.research.ibm.com/cell/
- **The Cell BE at IBM alphaWorks**
  - http://www.alphaworks.ibm.com/topics/cell
- **Cell BE at IBM Engineering & Technical Services**
  - http://www-03.ibm.com/technology/
- **IBM Power Architecture**
  - http://www-03.ibm.com/chips/power/
- **Cell BE documentation at IBM Microelectronics**
  - http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_EngineCell
- **Linux info at the Barcelona Supercomputing Center website**
  - http://www.bsc.es/projects/deepcomputing/linuxoncell/