Cell Broadband Engine Architecture

Leonardo Garcia

Staff Software Enginner

Linux Technology Center - IBM

2008-04-09 – Unicamp

Agenda – Dia 2

- Programação para Cell
  - Comunicação
    - Comandos do MFC
    - DMA
    - Mailboxes
  - SIMD
  - Análise de performance
  - Modelo de software
  - Técnicas de desenvolvimento de software
  - Onde encontrar mais informações

Comunicação no processador Cell

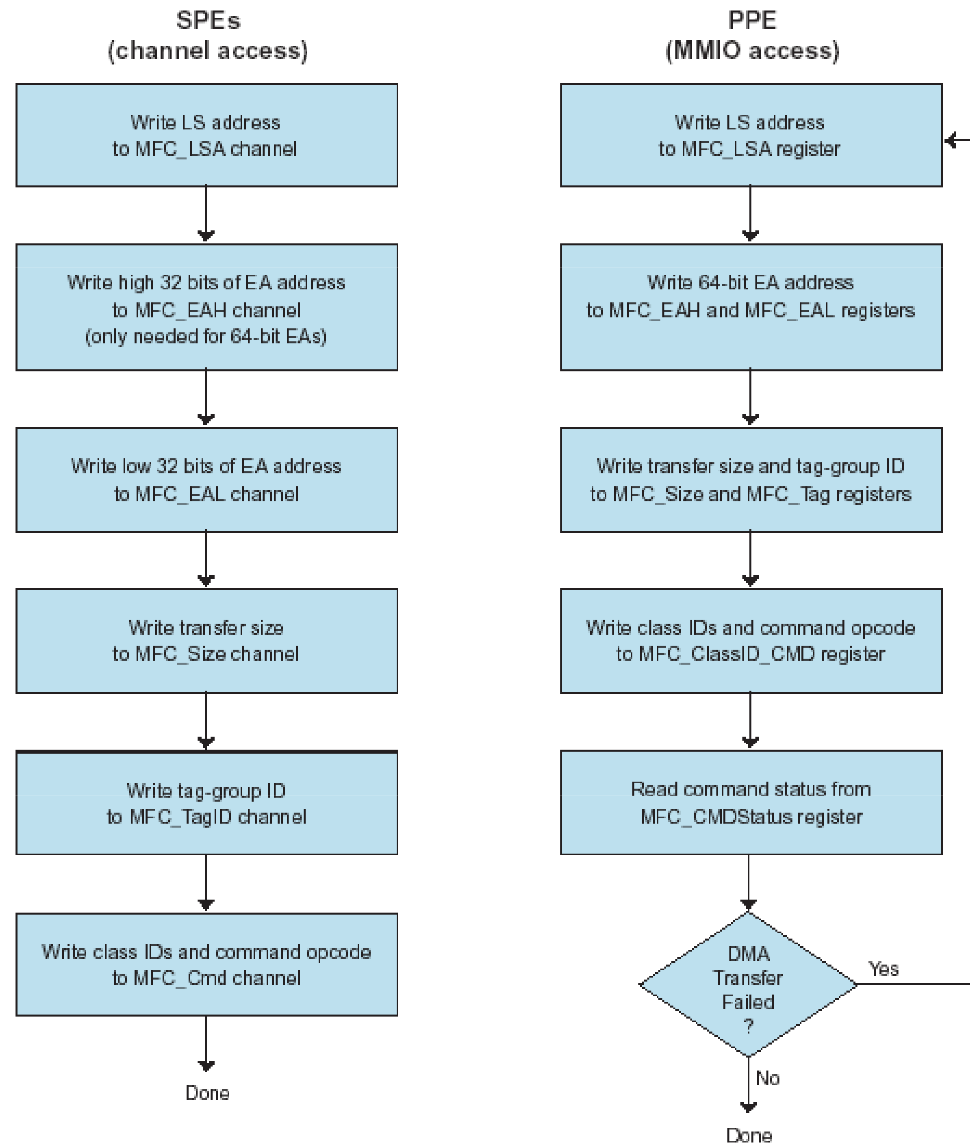
- DMA: move dados e instruções da memória principal para o Local Store e vice-versa. As transferências são assíncronas e são feitas em paralelo com o processamento da SPU.
- Mailboxes: comunicação de controle entre PPE, SPEs e outros dispositivos. Cada SPE tem duas caixas de saída e uma de entrada
- Sinais: comunicação de controle vinda do PPE ou de outros dispositivos.

Todos são controlados pelo MFC!

Comandos do MFC

- Podem ser gerados de duas formas:
  - Código rodando em um SPE executa uma série de escritas e/ou leituras no seu canal de instruções.
  - Código rodando no PPE ou em outros dispositivos faz uma série de stores e/ou loads em registradores do MFC mapeados em memória.
- Os comandos MFC são enfileirados em duas filas independentes:
  - MFC SPU Command Queue: para comandos associados à SPU iniciados pelo canal de controle.
  - MFC Proxy Command Queue: para comandos iniciados pelo PPE ou por outro dispositivo através dos registradores mapeados em memória.
- **Existem comandos MFC atômicos.**

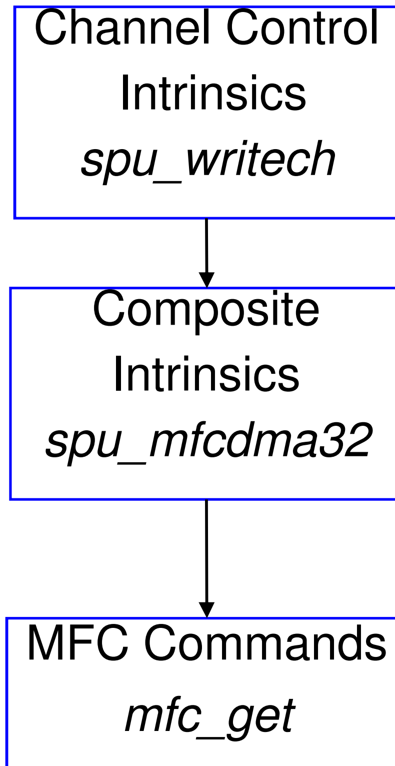
Seqüência para geração de comandos MFC



Comandos DMA

- São comandos MFC que transferem dados
- A direção da transferência é sempre em relação ao SPE:
  - Para o SPE (da memória principal para o Local Store): **get**
  - Do SPE (do Local Store para a memória principal): **put**

Comandos DMA



Macros definidas em

spu_mfcio.h

Comandos DMA

- DMA get (da memória principal para o Local Store):

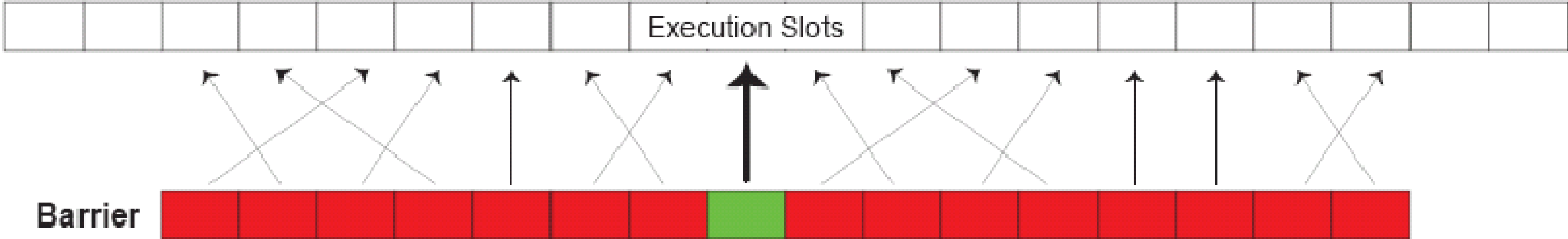
```
(void) mfc_get( volatile void *ls, uint64_t ea, uint32_t size,
    uint32_t tag, uint32_t tid, uint32_t rid)
```

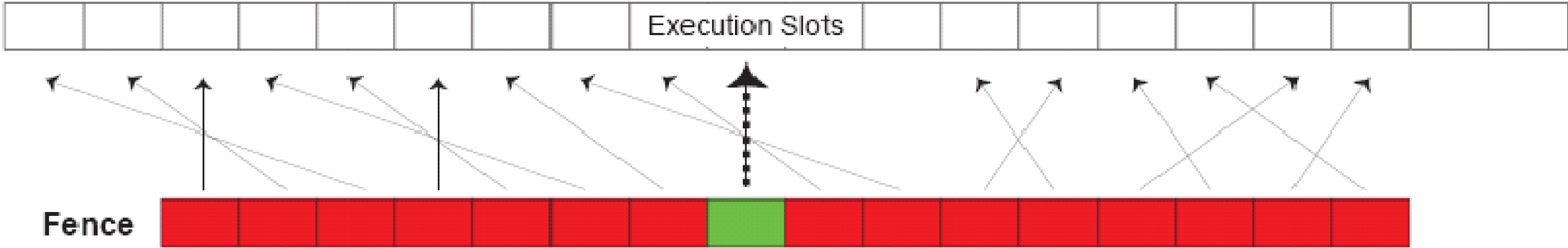
- DMA put (do Local Store para a memória principal):

```
(void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size,
    uint32_t tag, uint32_t tid, uint32_t rid)
```

- Os comandos não necessariamente são executados na ordem em que eles aparecem no programa.
- Para assegurar a ordem de execução das requisições de DMA:
  - mfc_putf : **fenced** (todos os comandos executados antes deste com o mesmo tag group devem terminar antes deste; comandos executados depois deste podem terminar antes dele)
  - mfc_putb : **barrier** (este comando e nenhum outro comando depois deste com o mesmo tag group será executado até que todos os comandos com este tag group executados anteriormente estejam completos)

Barriers e Fences





Earlier Instructions → Later Instructions
Time

Synchronizing command
Non-synchronizing command
Execution slot

DMA

- Transferências DMA
  - Podem ser de 1, 2, 4, 8, 16 ou múltiplos de 16 bytes, até 16 KB por transferência.
  - Alinhamento dos dados em 128 bytes é mais eficiente.
- Filas de comandos DMA por SPU
  - Fila de tamanho máximo 16 para transferências iniciadas pelo SPE
  - Fila de tamanho máximo 8 para transferências iniciadas pelo PPE
  - DMAs iniciados pelo SPE são preferíveis.
- DMA tags
  - Cada comando DMA tem um tag de 5 bits
  - O mesmo identificador pode ser usado em vários comandos
  - O tag pode ser usado para saber o status ou para se esperar a finalização de vários comandos DMA, já que os comandos DMA não são bloqueantes
- DMA lists
  - Um comando DMA pode executar uma lista de transferências
  - Função scatter-gather
  - Uma lista pode ter até 2K requisições

DMA Commands

- **Set tag mask**

```
unsigned int tag_mask;
mfc_write_tag_mask(tag_mask);
```

- **Fetch tag status**

```
unsigned int result;
result = mfc_read_tag_status();    /* or mfc_stat_tag_status(); */
```

  - É feito um AND lógico do tag status com a tag mask atual
  - Tag status bit '1' indica que não há nenhum DMA com aquele tag em execução ou na fila para ser executado.

- **Wait for any tagged DMA**

```
mfc_read_tag_status_any():
```

  - Espera até que um comando de alguma das tags marcadas se complete.

- **Wait for all tagged DMA**

```
mfc_read_tag_status_all():
```

  - Espera até que todos os comandos das tags marcadas completem-se.

DMA: lendo para o Local Store

```
inline void dma_mem_to_ls(unsigned int mem_addr,
         volatile void *ls_addr,unsigned int size)
{
  unsigned int tag = 0;
  unsigned int mask = 1;
  mfc_get(ls_addr,mem_addr,size,tag,0,0);
  mfc_write_tag_mask(mask);
  mfc_read_tag_status_all();
}
```

DMA: lendo para a memória principal

```
inline void dma_ls_to_mem(unsigned int mem_addr,volatile
void *ls_addr, unsigned int size)
{
  unsigned int tag = 0;
  unsigned int mask = 1;
  mfc_put(ls_addr,mem_addr,size,tag,0,0);
  mfc_write_tag_mask(mask);
  mfc_read_tag_status_all();
}
```

DMA entre SPEs

- É possível fazer transferências entre Local Stores. Para isso, o SPE que gera o comando DMA deve ter o endereço da Local Store do outro SPE mapeado em um endereço efetivo de 32 bits.
- O PPE consegue obter o endereço efetivo de um Local Store. Este valor pode ser enviado ao SPE por DMA ou mailbox posteriormente.

```
#include <libspe2.h>
spe_context_ptr_t speid;
void *spe_ls_addr;
..
spe_ls_addr = spe_ls_area_get(speid);
```

DMA

- A performance das DMAs são melhores quando o endereço de partida e de destino estão alinhados em 128 bits
- Uma linha de cache PPE tem 8 quadwords. Transferências que se iniciam ou terminam no meio de uma linha de cache fazem a transferência parcial da linha na primeiro ou última requisição ao barramento.

Mailboxes

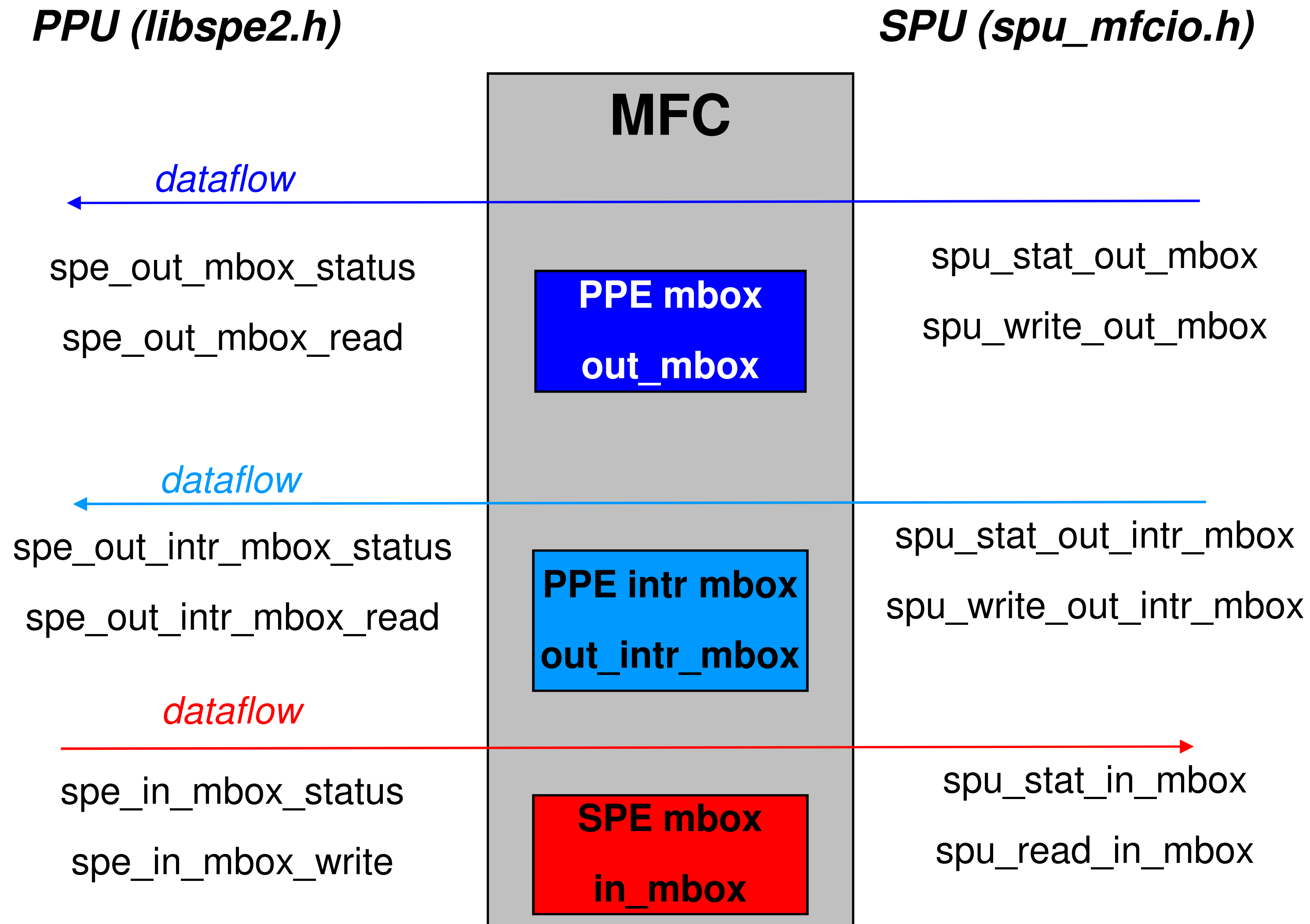
- Ideal para comunicação de dados de até 32 bits, por exemplo, status gerais do programa.
- Pode ser usada entre PPE e SPE, entre SPEs e entre PPE e outros dipositivos ou entre SPE e outro dispositivo.

Mailboxes

Cada MFC provê três filas de mailboxes de 32 bits cada:

- PPE (SPU write outbound) mailbox queue
  - SPE escreve, PPE lê.
  - 1 entrada
  - SPE bloqueia se escrever para este mailbox e ele estiver cheio. Para evitar isso a SPU pode verificar se a mailbox está vazia antes de fazer a escrita.
  - A PPU deve verificar se tem alguma coisa para ler antes de ler este mailbox pois senão pode ler lixo.
- PPE (SPU write outbound) interrupt mailbox queue
  - Como a PPE mailbox queue, mas envia uma interrupção à PPE quando é escrita
- SPE (SPU read inbound) mailbox queue
  - PPE escreve, SPE lê.
  - 4 entradas
  - Pode ser sobrescrita se uma PPE escrever na mailbox e a fila estiver cheia
  - Se a SPU ler uma mailbox que não tenha nada para ser lido, ela é bloqueada

Mailbox API



DMA e Mailbox - PPU

```
typedef struct {
  char * memory;
  int first;
  int last;
  void * lstore[8];
  char padding[84];
} control_block;
```

```
#include <stdio.h>
#include <libspe2.h>
#include <libmisc.h>
#include <pthread.h>
#include <string.h>
#include "control.h"

extern spe_program_handle_t hello_spu;
char buffer[128] __attribute__ ((aligned(128)));

#define ACTIVE_SPUS 6

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

        spe_context_run(*((spe_context_ptr_t *)arg),
&entry, 0, NULL, NULL, NULL) ;
        pthread_exit(NULL);
}

int main()
{

        int i;
        spe_context_ptr_t ctxs[ACTIVE_SPUS];
        pthread_t threads[ACTIVE_SPUS];

        control_block * cb = (control_block
*)malloc_align(128,7);
        cb->first = 0;
        cb->last = ACTIVE_SPUS - 1;
        cb->memory = buffer;
        strcpy (buffer, "Zao shang hao!");
```

```
 for (i=0;i<ACTIVE_SPUS;i++){
                ctxs[i] = spe_context_create (0,
NULL);
                spe_program_load (ctxs[i],
&hello_spu);
                pthread_create (&threads[i], NULL,
&ppu_pthread_function, &ctxs[i]);
        }
        for (i=0;i<ACTIVE_SPUS;i++){
                cb->lstore[i] =
spe_ls_area_get(ctxs[i]);
                while (!spe_out_mbox_status(ctxs[i]))
{}
                unsigned int temp;
                spe_out_mbox_read(ctxs[i], &temp, 1);
                cb->lstore[i] += temp;

        }
        for (i=0;i<ACTIVE_SPUS;i++){
                unsigned int data;
                data = (unsigned int)(cb);
                spe_in_mbox_write(ctxs[i], &data, 1,
SPE_MBOX_ANY_NONBLOCKING);
        }

        for (i=0;i<ACTIVE_SPUS;i++){
                spe_in_mbox_write(ctxs[i], (unsigned
int *)&i, 1, SPE_MBOX_ANY_NONBLOCKING);
                pthread_join (threads[i], NULL);
        }

        printf("PPE says %s\n", buffer);
        return 0;
}
```

DMA e Mailbox - SPU

```
#include <stdio.h>
#include <spu_mfcio.h>
#include <malloc_align.h>
#include <string.h>
#include "../control.h"

int main()
{
  int myId;
  char * buffer = malloc_align(128,7);
  control_block * ls_cb =
malloc_align(sizeof(control_block),7);
  control_block * cbPtr;

  // Send buffer offset
  spu_write_out_mbox((unsigned int)buffer);

  // Get control block address and transfer it to local
storage
  cbPtr = (control_block *) spu_read_in_mbox();
  mfc_get (ls_cb, (unsigned int)cbPtr, sizeof(control_block),
1, 0, 0);
  mfc_write_tag_mask(1<<1);
  mfc_read_tag_status_all();

  // Wait to execute until it is our turn
  myId = spu_read_in_mbox();

  // First SPU in the chain reads from PPU memory
  if (myId == ls_cb->first) {
    mfc_get(buffer, (unsigned int)ls_cb->memory, 128, 1, 0,
0);
    mfc_write_tag_mask(1<<1);
    mfc_read_tag_status_all();
  }

  printf ("SPE %d says %s\n", myId, buffer);
```

```
 switch ( myId ) {
    case 0: { strcpy(buffer,"Dobry dien!"); break; }
    case 1: { strcpy(buffer,"Bon giorno!"); break; }
    case 2: { strcpy(buffer,"Buenas dias!"); break;
}
    case 3: { strcpy(buffer,"Ohayo gozaimasu!");
break; }
    case 4: { strcpy(buffer,"Boker tov!"); break; }
    case 5: { strcpy(buffer,"Dobre jitro!"); break;
}
    case 6: { strcpy(buffer,"Bon jour!"); break; }
    case 7: { strcpy(buffer,"Chao buoi sang!");
break; }
  }

  // Send message to next SPU (or PPU if we are the
last SPU)
  if (myId == ls_cb->last) {
    mfc_put(buffer, (unsigned int)ls_cb->memory,
128, 1, 0, 0);
  }
  else {
    mfc_put(buffer, (unsigned int)ls_cb-
>lstore[myId+1], 128, 1, 0, 0);
  }

  mfc_write_tag_mask(1<<1);
  mfc_read_tag_status_all();

  return 0;
}
```

SIMD

- PPU tem VMX
- SPU é uma arquitetura instrinsicamente vetorial
- Ambos suportam operandos de 128 bits
  - 4 fullwords
  - 8 halfwords
  - 16 bytes
  - A SPU ainda suporta 2 doublewords
- A mesma operação é aplicada sobre todos os elementos do vetor ao mesmo tempo.

SIMD

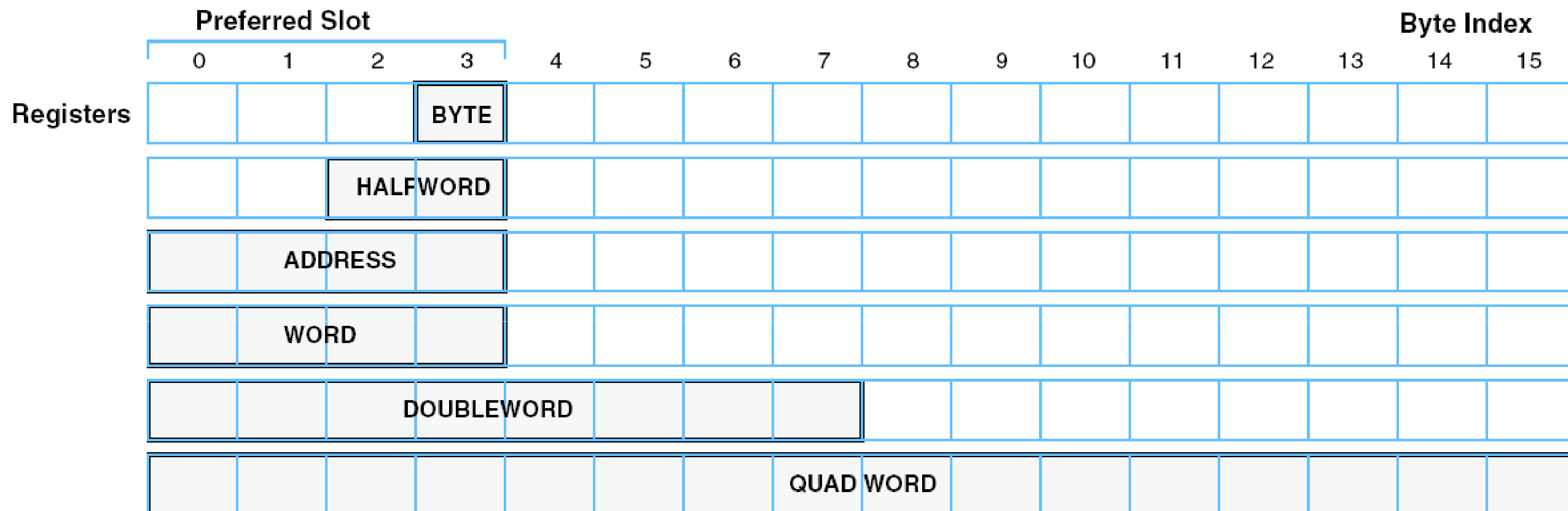
- Existem basicamente duas opções de programação SIMD:
  - Vetorização de código feita pelo programador
  - Auto-vetorização aplicada sobre um código tradicional (suporte limitado no XL C/C++)
- Ponteiros de vetores
  - Exemplo: vector float * p
  - p+1 aponta para o próximo vetor (offset de 16 bytes)
  - É possível fazer cast entre tipos vetoriais e escalares

Arquitetura SIMD dos SPEs

- Processamento escalar é possível mas não aproveita toda a eficiência do processador:
  - Mesmo o processamento escalar é feito com instruções vetoriais operando sobre vetores
  - Operações escalares são definidas por como se usa a instrução: não existem opcodes escalares na SPU
  - Argumentos escalares de instruções são executados de maneira mais eficiente se estiverem nos “preferred slots”. No entanto, a computação é feita em qualquer slot.

Register Scalar Data Layout

- Slots preferidos nos bytes de 0 a 3
  - São usadas por instruções que trabalham com dados escalares, por exemplo, branches



Promoting Scalar Data Types to Vector Data Types

- SPU faz loads e stores de uma quadword por vez
- Valores de operandos escalares (incluindo endereços) são guardados no preferred slot SIMD de um registrador
- Loads e stores de valores escalares requerem várias instruções para formatar o dado para uso no SPE
- Estratégias para operar com valores escalares de maneira mais eficiente:
  - Mude os tipos escalares para vetores para eliminar três instruções associadas a cada load e store de valores escalares
  - Agrupe escalares em grupos de 128 bits e extraia-os manualmente dos grupos de acordo com sua necessidade.

Intrinsics for Changing Scalar and Vector Data Types

Instruction	Description
d = spu_insert	Insert a scalar into a specified vector element.
d = spu_promote	Promote a scalar to a vector.
d = spu_extract	Extract a vector element from its vector.

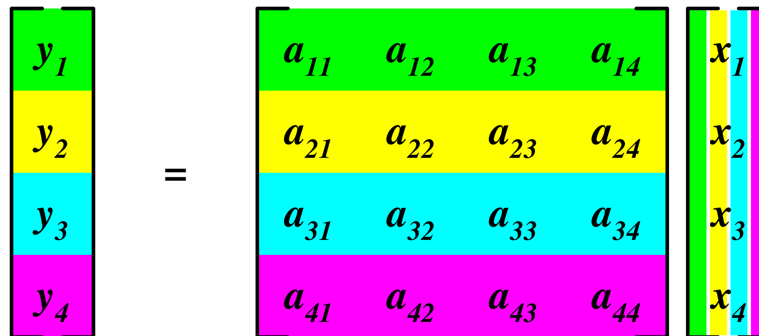
Exemplo SIMD: multiplicação de matrizes

- Este é um problema que também pode ser paralelizado. No entanto, é importante vermos também que talvez apenas sua vetorização seja mais vantajosa do que sua paralelização.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

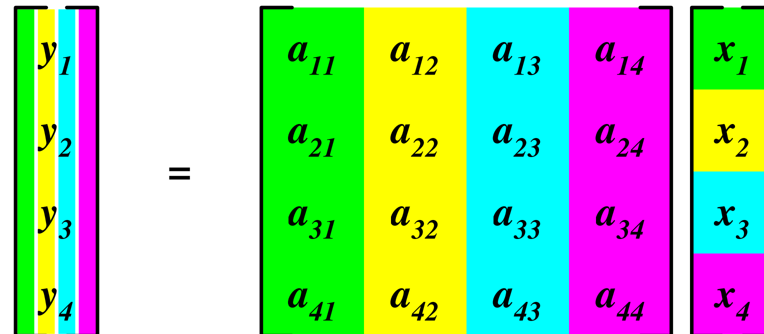
- Duas possíveis soluções:
  - Produto escalar: cada linha pelo elemento do vetor (linha da outra matriz)
  - Soma de vetores: cada coluna multiplicada por um elemento do vetor

Exemplo SIMD: multiplicação de matrizes



- Cada linha é multiplicada pelo vetor x
- Efetua-se a redução do vetor calculado: soma dos quatro valores do vetor
- Coloca o resultado na posição correta do vetor resultado

Exemplo SIMD: multiplicação de matrizes



- Copie cada elemento de x em todas as posições de um vetor (splat)
- Multiplique a coluna pelo vetor “splated” e a adicione com o vetor resultado (multiply-and-add)

SIMD tradeoffs

§ A técnica de vetorização escolhida dependerá de vários fatores:
- Organização dos vetores de dados, por exemplo: canais ARGB e operações com números imaginários.
- Qual o instruction set disponível (algumas operações SPU não existem em VMX e vice-versa)
- Oportunidades de loop unrolling
- Pipeline latencies (problemas com doubleword)

SIMD

```
#include <stdio.h>

int mult1(float *in1, float *in2, float *out, int N)
{
   int i;
   vector float *a = (vector float *) in1;
   vector float *b = (vector float *) in2;
   vector float *c = (vector float *) out;

   int Nv = N/4;

   for (i=0;i<Nv;i++)
   {
         c[i] = spu_mul(a[i], b[i]);
   }
   return 0;
}
```

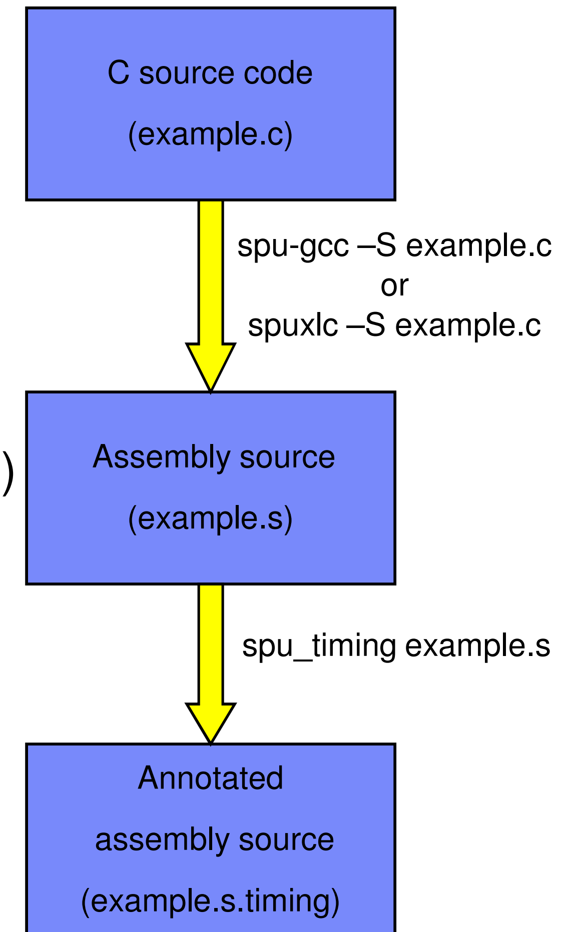
```
#include <stdio.h>
#define N 16
int mult1(float *in1, float *in2, float *out, int
   num);
float a[N] __attribute__ ((aligned(16)))
     = { 1.1, 2.2, 4.4, 5.5,
         6.6, 7.7, 8.8, 9.9,
         2.2, 3.3, 3.3, 2.2,
         5.5, 6.6, 6.6, 5.5};
float b[N]  __attribute__ ((aligned(16)))
     = { 1.1, 2.2, 4.4, 5.5,
         5.5, 6.6, 6.6, 5.5,
         2.2, 3.3, 3.3, 2.2,
         6.6, 7.7, 8.8, 9.9};
float c[N] __attribute__ ((aligned(16)));

int main()
{
   int num = N;
   int i;

   mult1(a, b, c, num);

   for (i=0;i<N;i+=4)
         printf("%.2f %.2f %.2f %.2f\n", c[i],
   c[i+1], c[i+2], c[i+3]);

   return 0;
}
```

SPE Performance Tools

- Para se tirar o máximo da programação vetorial, o SPE possui várias ferramentas de análise de performance disponíveis
- Análise estática (spu_timing)
  - Mostra o código assembly com os estados do pipeline
- Análise dinâmica
  - CBE System Simulator
  - FDPR-PRO (Feedback Directed Program Restructuring)
    - Ferramenta de otimização que modifica os executáveis do programa através da sua análise de execução após a linkagem final

SPU Timing

- Análise estática do código assembly
- Não leva em conta
  - Instruction fetch stall
  - Local Store contention
  - Branching: assume execução linear (branchless)



SPU Timing

```
000000 0D 01
000000 1D 0123
000001 0d  12
000002 1d  -2345
000003 0D    3
000003 1D    3456
000004 0D     45
000004 1D     456789
000005 0D      56
000005 1D      5

000006 0d       67
000007 1d       -789012
000008 1          890123
000014 0           -----456789
000020 1                 -----012345

000022 1                        2345
000023 1                         3456
```

```
        .file     "example.c“
.text
        .align    3
        .global   saxpy
        .type     saxpy, @function
saxpy:
        ila       $2,66051
        shlqbyi   $7,$3,0
        cgti      $3,$3,0
        shufb     $8,$4,$4,$2
        nop       $127
        biz       $3,$lr
        ori       $4,$7,0
        hbra      .L8,.L4
        il        $7,0
        lnop
.L4:
        ai        $4,$4,-1
        lqx       $2,$7,$5
        lqx       $3,$7,$6
        fma       $2,$8,$2,$3
        stqx      $2,$7,$6
.L8:
        brnz      $4,.L4
        bi        $lr
```

IBM Systemsim Cell Simulator

- Permite o acesso aos dados de execução do processador (conteúdo da memória, registradores PPU e SPU, Local Storage SPU, etc.)
- Possui o modelo lógico da execução na PPU.
- Possui o modelo lógico-temporal da execução nas SPUs e do funcionamento do sistema de memória.
- Gera dados estatísticos da execução do SPE na parte do código com o profile ligado
  - Ciclos, instruções, e CPI
  - Single/Dual issue rates
  - Stall statistics
  - Uso dos registradores
  - Histograma de instruções
  - Branch misprediction rate
  - etc.

IBM Systemsim Cell Simulator profile

```
#include “profile.h”

prof_clear();    // clear performance info
prof_start();  // start recording performance info

< something interesting >

prof_stop();  // stop recording performance info
```

IBM Systemsim Cell Simulator profile

```
SPU DD3.0
***
Total Cycle count              22082564
Total Instruction count        2583628
Total CPI                      8.55
***
Performance Cycle count        6153081
Performance Instruction count  2573522 (2572961)
Performance CPI                2.39 (2.39)

Branch instructions            143160
Branch taken                   142880
Branch not taken               280

Hint instructions              140
Hint hit                       142740

Contention at LS between Load/Store and Prefetch 142880

Single cycle                                       1715121 ( 27.9%)
Dual cycle                                          428920 (  7.0%)
Nop cycle                                                0 (  0.0%)
Stall due to branch miss                              7560 (  0.1%)
Stall due to prefetch miss                               0 (  0.0%)
Stall due to dependency                            4001060 ( 65.0%)
Stall due to fp resource conflict                        0 (  0.0%)
Stall due to waiting for hint target                   420 (  0.0%)
Issue stalls due to pipe hazards                         0 (  0.0%)
Channel stall cycle                                      0 (  0.0%)
SPU Initialization cycle                                 0 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                        6153081 (100.0%)

Stall cycles due to dependency on each pipelines
 FX2        143020 (  3.6% of all dependency stalls)
 SHUF       857560 ( 21.4% of all dependency stalls)
 FX3        0 (  0.0% of all dependency stalls)
 LS         857280 ( 21.4% of all dependency stalls)
 BR         0 (  0.0% of all dependency stalls)
 SPR        0 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        2143200 ( 53.6% of all dependency stalls)
 FP7        0 (  0.0% of all dependency stalls)
 FPD        0 (  0.0% of all dependency stalls)

The number of used registers are 15, the used ratio is 11.72
```

Estatísticas totalizadas para a execução da SPU não afetadas pelo profile

Estatísticas totalizadas para a região em que se especificou fazer o profile

IBM Systemsim Cell Simulator profile: histrograma

```
 mnemonic
----------+---------
      lnop       281
      hbra       140
         a    142880
       and       141
        ai    428640
       brz    143020
      stqx    285760
       lqa    142880
        br       140
     fsmbi       140
       lqx    571520
    rotqby    142880
       nop       140
        il       280
       ila       140
      cgti       140
        fm    142880
       ceq    142880
     shufb    142880
       fma    285760
```

IBM Systemsim Cell Simulator profile: branch history

```
Branch histories
INST     ADDRESS:       count      taken  not_taken       hint        hit
-------------------------------------------------------------------------
brsl     0x00014:           1          1          0          0          0
brnz     0x0017c:          15         14          1          1         14
brnz     0x001c4:           1          0          1          0          0
brnz     0x0026c:           1          0          1          0          0
bi       0x00274:           1          1          0          0          0
br       0x003c4:      142881     142741        140        141     142741
sync     0x004e0:           1          0          1          0          0
bi       0x004ec:           1          1          0          0          0
brnz     0x0004c:          28         27          1          0          0
stop     0x00090:           1          0          1          0          0
br       0x0009c:          32         32          0          0          0
bi       0x3fe14:           1          1          0          0          0
brnz     0x000d4:           1          1          0          0          0
brsl     0x000fc:           1          1          0          0          0
brz      0x00364:         140          0        140          0          0
br       0x003cc:         140        140          0          0          0
-------------------------------------------------------------------------
Total                  143246     142960        286        142     142755
```

IBM Systemsim Cell Simulator profile: hint statistics

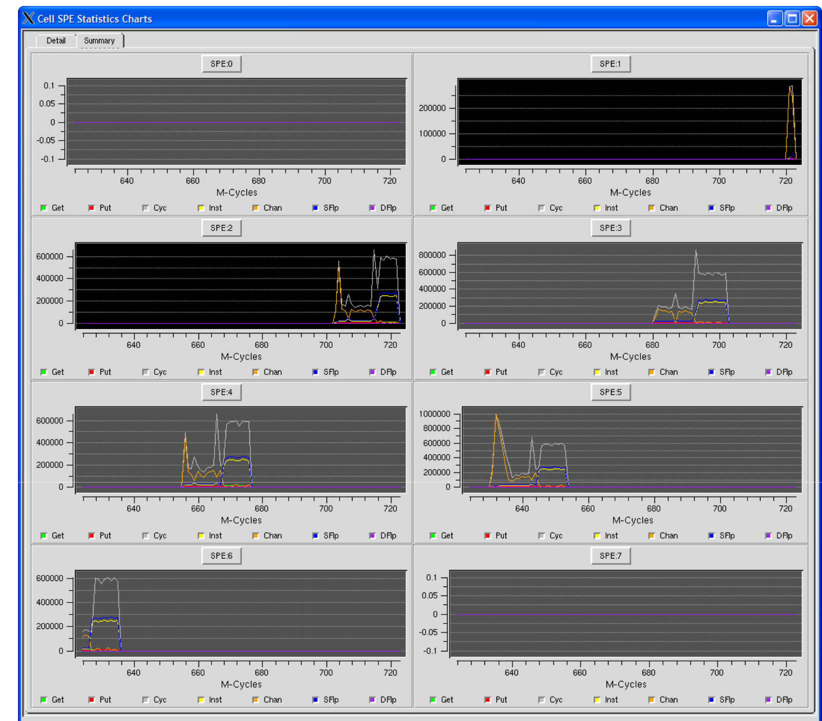
```
Hint histories
INST     ADDRESS: br_addr  tgt_addr         count           hit
-----------------------------------------------------------------
hbra     0x0011c: 0x0017c  0x00140              1            14
hbra     0x00374: 0x003c4  0x004e0              1             1
hbra     0x0036c: 0x003c4  0x00380            140        142740
-----------------------------------------------------------------
Total                                         142        142755
```

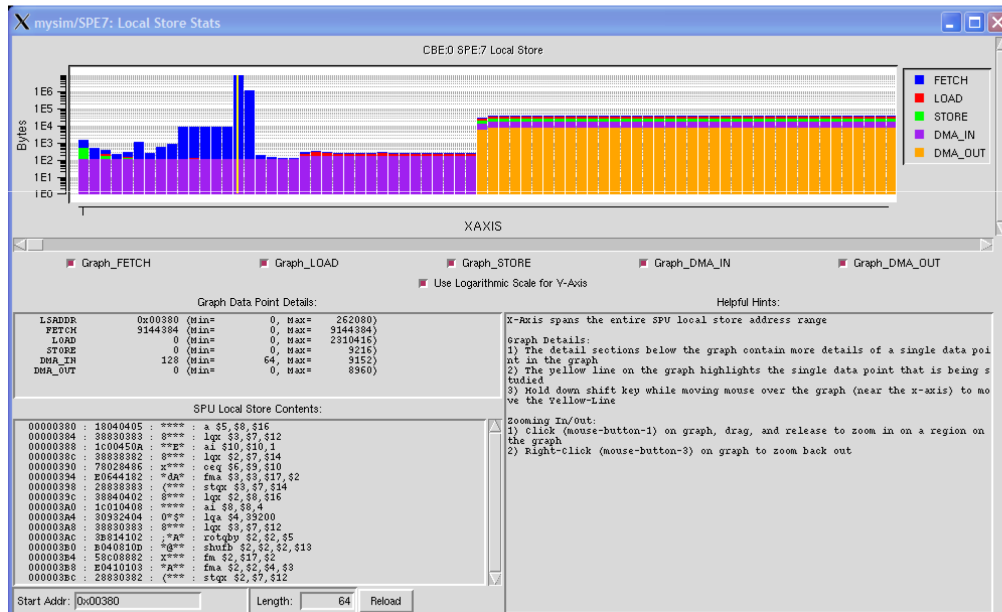
IBM Systemsim Cell Simulator profile: register utilization

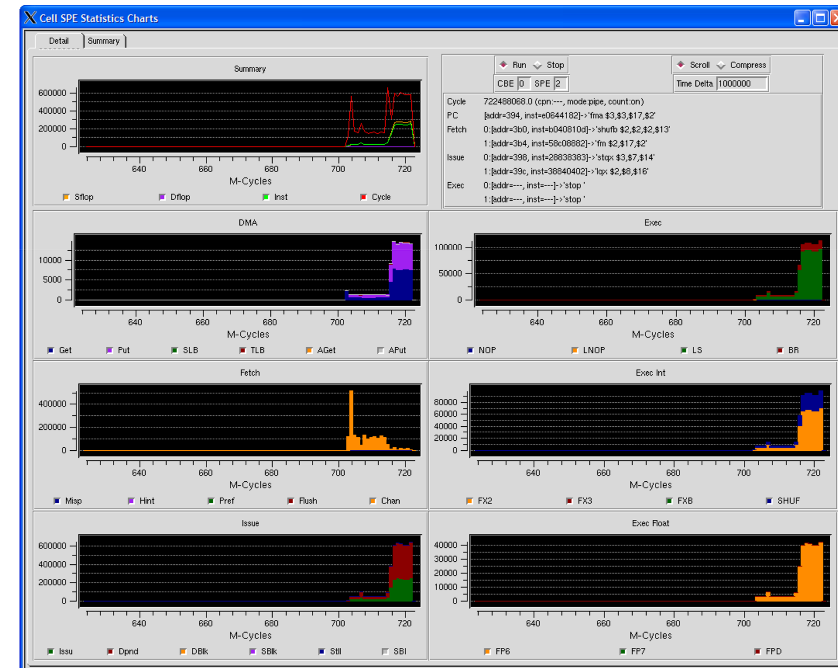
Register use:	READS(%tot)	WRITES(%tot)	R+W(%tot)
2:	1000300(14.00)	857420(12.00)	1857720(26.00)
3:	428640(6.00)	428640(6.00)	857280(12.00)
4:	142880(2.00)	142880(2.00)	285760(4.00)
5:	142880(2.00)	142880(2.00)	285760(4.00)
6:	142880(2.00)	142880(2.00)	285760(4.00)
7:	857280(12.00)	143020(2.00)	1000300(14.00)
8:	428640(6.00)	143020(2.00)	571660(8.00)
9:	143020(2.00)	0(0.00)	143020(2.00)
10:	285760(4.00)	143020(2.00)	428780(6.00)
12:	428640(6.00)	0(0.00)	428640(6.00)
13:	142880(2.00)	140(0.00)	143020(2.00)
14:	285760(4.00)	0(0.00)	285760(4.00)
16:	285760(4.00)	0(0.00)	285760(4.00)
17:	285760(4.00)	0(0.00)	285760(4.00)
31:	282(0.00)	141(0.00)	423(0.01)
TOTAL use:	5001362(69.99)	2144041(30.01)	7145403

IBM Systemsim Cell Simulator

- Também possui ferramentas gráficas de visualização das estatísticas







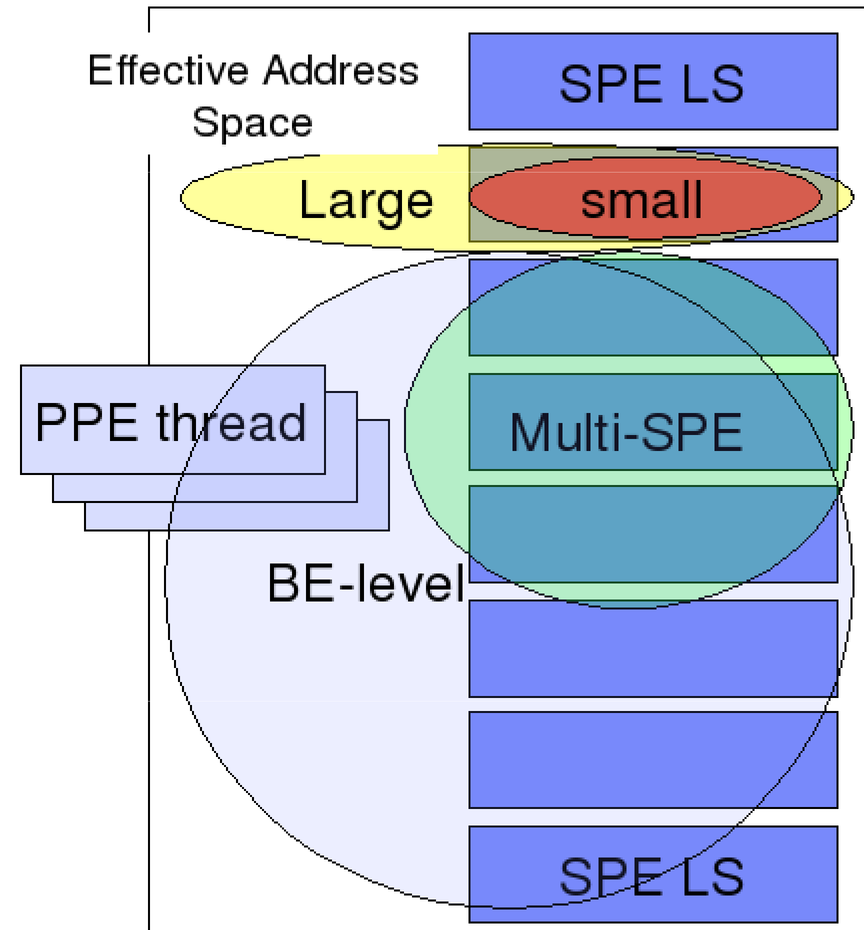
Um lembrete: considerações sobre branches SPE

- Mispredicts custa 18 ciclos
- Técnicas para se evitar branches
  - Escrever o caminho mais freqüente como código inline
  - Calcular os dois caminhos do branch e só depois escolher o resultado
  - Unroll loops: também diminui os dependency stalls
  - Use as instruções de hint

Modelos de programação para Cell

Single Cell environment:

- **PPE programming models**
- **SPE Programming models**
  - Small single-SPE models
  - Large single-SPE models
  - Multi-SPE parallel programming models
- **Cell Embedded SPE Object Format (CESOF)**
- **Multi-tasking SPEs**
  - Local Store resident multi-tasking
  - Self-managed multi-tasking
  - Kernel-managed SPE scheduling and virtualization



Modelos de programação para PPE

- O PPE é um núcelo PowerPC de 64 bits e, por isso, herda os modelos de programação tradicionais.
- Cell: o PPE funciona como um controlador ou facilitador
  - CESOF provê suporte ao manuseio de imagens SPE durante a execução de aplicações PPE
  - O programa PPE estabelece um ambiente de execução para as aplicações SPE
    - Mapeamento do Local Store em memória, tratamento de exceções, controle de execução do SPE
  - O programa PPE é capaz de fazer scheduling de aplicações SPE e controlar o hypervisor do Cell BE
  - Provê acesso à chamadas de sistema a partir dos SPEs
    - Printf, I/O, etc.

Modelos de programação para SPE

- **Small single SPE programming model**
  - Uma thread SPE apenas
  - Usa no máximo os 256 KB existentes no Local Store
  - É suficiente para muitos casos
  - Endereçamento SPE disjunto do PPE
  - Entrada e saída explicita no programa SPE
    - Argumentos recebidos na criação da thread SPE
    - DMA
    - Mailboxes
    - Chamadas de sistema a partir do SPE

Modelos de programação para SPE

- **Large single-SPE programming model**
  - Código e dados não cabem no Local Store
  - O PPE, através do kernel e da biblioteca de runtime do SPE mapeia a memória principal como se fosse uma memória secundária do SPE.
  - O SPE acessa esta memória secundária através de seus mecanismos de DMA
  - Lembre-se: as latências de DMA podem ser críticas para o desempenho de um programa SPE que faz várias movimentações de dados. Pre-fetch de dados esconde esta latência: **double-buffering** ou **multi-buffering**.

Modelos de programação para SPE

- **Large single-SPE programming model**
  - Pode-se usar a memória secundária para
    - Buscar/guardar dados que serão processados
    - Configurar um ambiente em que o Local Store funcionaria como um cache de dados (existe uma biblioteca para gerenciamento de cache em software disponibilizada junto com o Cell SDK)
    - Guardar instruções (**suporte a overlay nas ferramentas disponibilizadas no Cell SDK**)
    - Guardar código e dados empacotados como job que são enviados para serem executados por um núcleo de software SPU comum (**modelo multi-task**)

Modelos de progamação para Cell

- **Cell Embedded SPE Object Format (CESOF)**
- Permite a resolução de referências ao espaço de endereçamento global pelo SPE
- Torna possível a transferência de dados dinâmicamente entre o PPE e os SPEs e entre estes e dispositivos externos.

Modelos de programação paralelos

- Modelos de programação paralelo tradicionais são aplicáveis
- Baseados na interação de programas single-SPE
- Utilizam os mecânismos de comunicação do SPE
  - Comandos MFC de movimentação atômica de dados
  - Mailboxes
  - Sinais
  - Eventos e interrupções

Modelos de programação paralelos

- **Memória compartilhada**
  - Cell pode ser programado como um multi-processador com memória compartilhada
  - CESOF suporta o uso de variáveis compartilhadas através da sua capacidade de acessar endereços no espaço de endereçamento da memória principal
  - Uso de locks!
  - Existem frameworks que ativam o compartilhamento de regiões da memória entre PPE e SPE

Modelos de programação paralelos

- **Streaming**
  - Uma grande quantidade de dados é processada simultaneamente por um grupo de SPUs. Cada SPU cuida do processamento de uma parte dos dados.
  - As SPUs iniciam os DMAs e requisitam mais dados ao PPE, que se encarrega de gerenciar que parte dos dados será enviada cada vez.

Modelos de programação paralelos

- **Pipeline**
  - Os dados são transferidos entre as SPUs (de Local Store para Local Store, sem acesso à memória principal para diminuir o gargalo)
  - Cada SPU processa uma parte do trabalho total a ser feita em cima dos dados
  - Facilita a implementação paralela de algoritmos que podem ser executados em pipeline
  - Load-balance é mais difícil

Modelos de programação paralela

- **Multi-tasking SPE – LS resident multi-tasking**
  - Cada SPE só pode rodar uma thread por vez.
  - Várias tarefas são carregadas no SPE e um controlador de qual tarefa será executada cada vez roda dentro da SPE
  - Sem proteção de memória entre as tarefas

Modelos de progamação paralela

- **Multi-tasking SPE – Self-managed multi-tasking**
  - Usa a memória principal para guardar as tarefas.
  - O contexto das tarefas é trocado por um pequeno núcleo de software rodando na SPE
  - Na troca, o contexto é guardado na memória principal até que ele seja escalonado novamente
  - Desvantagem: o SPE foi pensado para rodar suas tarefas até elas serem completadas

Modelos de programação paralela

- **Multi-tasking SPE – Kernel Managed**
  - Parecido com o Self-managed multi-tasking SPE, porém gerenciado pelo kernel rodando no PPE.

Notas:

- Modelos de programação podem ser misturados
- Talvez uma determinada aplicação precise de um modelo de programação diferente
- Um modelo de programação correto diminui o tempo de desenvolvimento e dá melhores resultados

Modelos de programação paralelos

- **Passagem de mensagens**
  - É seqüencial por natureza
  - Indicado apenas para casos em que o código mais os dados cabem no Local Store
  - Mais rápido que o modelo de memória compartilhada

Cell Software design

- Dois níveis de paralelismo
  - Dados vetoriais: SIMD
  - Tarefas executadas em paralelo
- Computação
  - Os 8 SPEs e o PPE possuem instruções vetoriais (SIMD)
  - As tarefas paralelizáveis devem ser distribuídas nos 8 SPEs e no PPE.
  - 256 KB Local Store por SPE para dados e código!
- Comunicação
  - DMA e banda de barramento
    - Granularidade do DMA: 128 bytes (acessos menores que este são ineficientes)
  - Controle de trafégo
    - Deve-se explorar a grande capacidade de processamento dos SPEs e a localidade de dados inerente a eles para diminuir o trafégo de dados necessário
  - Abstrações de shared memory e passagens de mensagens implicam em overheads
  - Sincronização de dados e DMA possuem latências indesejáveis

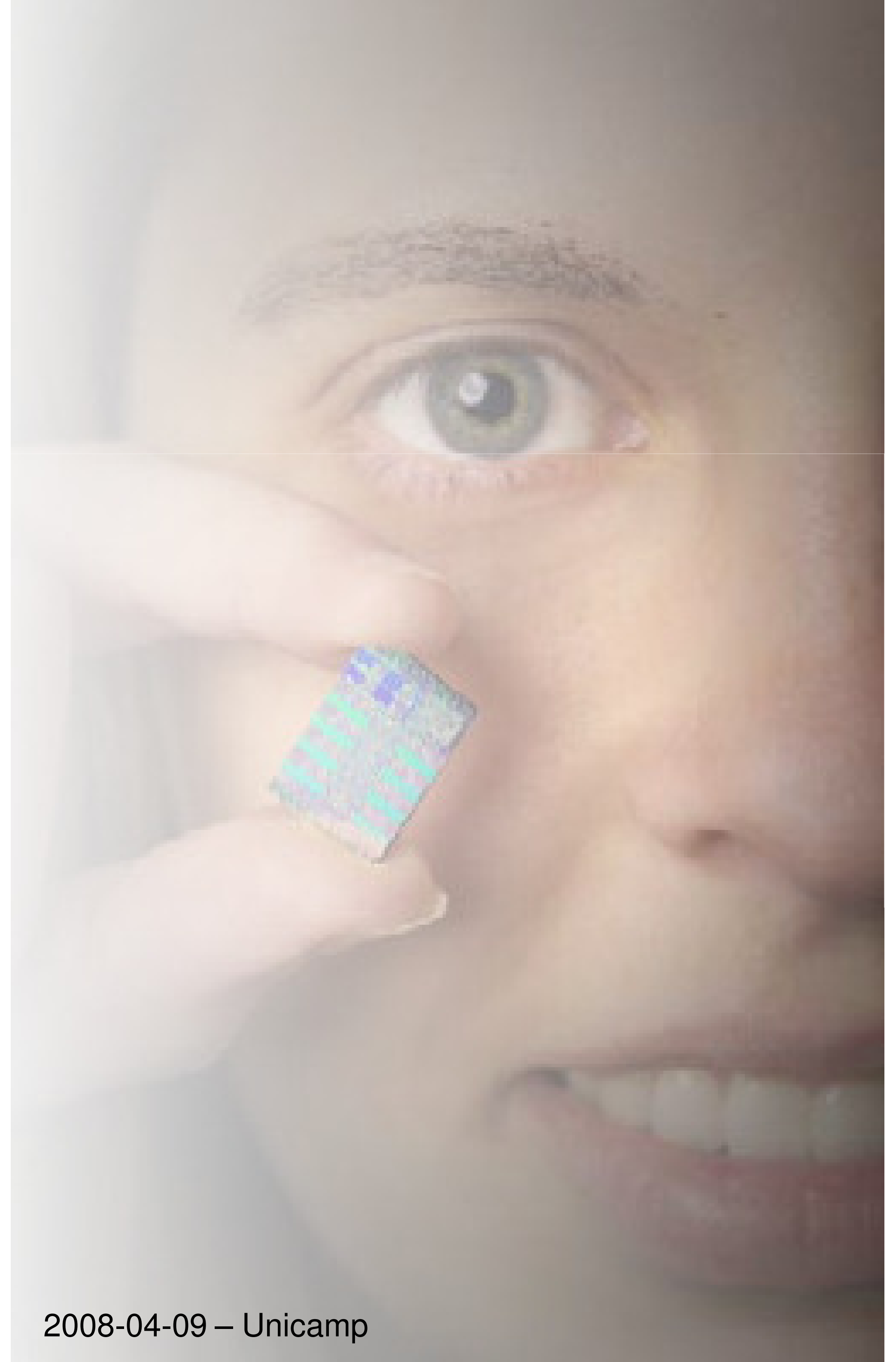
Fluxo de desenvolvimento de aplicações para Cell

- Estudo da complexidade do algoritmo
- Análise do fluxo dos dados, de seu layout e sua localidade
- Particionamento e mapeamento do algoritmo e estruturação do programa na arquitetura
- Desenvolvimento do controle PPE e do código escalar PPE
- Desenvolvimento do controle PPE e do particionamento do código SPE ainda escalar
  - Comunicação, sincronização, latências
- Transformar código escalar SPE em código SPE SIMD
- Rebalanceamento da computação com a movimentação de dados no SPE
- Outras otimizações
  - PPE SIMD, análise de gargalos e balanceamento
- **Importante: todas as tarefas de computação devem ser delegadas aos SPEs**
- **Importante: as funcionalidades providas pela microarquitetura não estão disponíveis apenas ao compilador mas aos usuários também: aproveite-as!**

Maiores informações

- IBM DeveloperWorks
- Barcelona Supercomputing Center
- IBM Academic Initiative
- IBM Students Portal
- Existem toneladas de documentação a respeito da Cell Broadband Engine Architecture na Internet

Perguntas?



Obrigado!

Leonardo Garcia
lagarcia@br.ibm.com
Staff Software Enginner
Linux Technology Center - IBM