Programming the Cell BE

Part 2

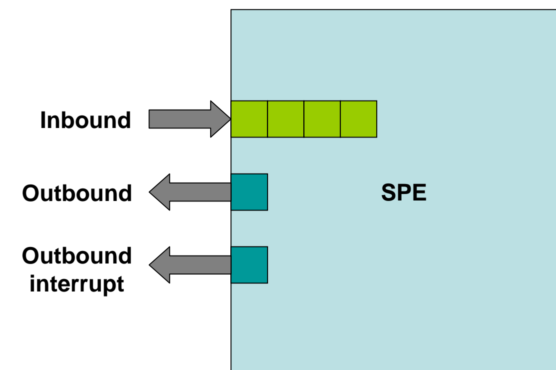
Aim of this lecture

- The aim of this lecture is to...
  - ...get familiar with the Cell Broadband Engine,
  - ...understand SIMD concepts and programming,
  - ...learn common loop optimizations,
  - ...learn how to use the QS21 Cell blades at HPC2N,
  - ...introduce Assignment 4.
- The aim of this lecture is to...
  - ...introduce the MFC and communication,
  - ...learn how to use a static performance analysis tool,
  - ...introduce the SoA / AoS storage formats,
  - ...discuss programming models.

Mailboxes

Small unidirectional communication

Mailboxes



Mailboxes

- Mailboxes are used for uni-directional communication of 32-bit unsigned data.
- Each SPE has 2 outgoing mailboxes and 1 incoming
  - Incoming:
    - SPE Inbound mailbox (4 entries)
  - Outgoing:
    - SPE Outgoing mailbox (1 entry)
    - SPE Outgoing interrupt mailbox (1 entry)
- Mailboxes can be read and written via intrinsics.
- Useful for instructing SPU what to do
  - Send application op-code for example (RPC implementation)
- No alignment restrictions

Using Mailboxes

```
// On the PPE (writing to SPE)
unsigned int data = 123;
spe_in_mbox_read(spe_ctxt, &data, 1, SPE_MBOX_ALL_BLOCKING);

// On the PPE (reading from SPE)
while( !spe_out_mbox_status(spe_ctxt) )
   ; // Busy wait
spe_out_mbox_read(spe_ctxt, &data, 1);



// On the SPE (reading)
unsigned int data;
data = spu_read_in_mbox();

// On the SPE (writing)
spu_write_out_mbox(data);
```

DMA Transfers

Asynchronous bulk transfers

DMA

- SPEs access main memory via an explicitly managed DMA engine that runs independently of the SPU (the MFC).
- Two directions:
  - get (memory to Local Store)
  - put (Local Store to memory)
- Two forms:
  - simple
  - list (suffix: l)

DMA Tag groups and synchronization

- Each DMA transfer can be tagged with a tag from 0 to 31
- MFC commands can be issued to wait for any or all of any specified set of tags (set is referred to as tag mask)
  - `mfc_write_tag_mask(tag_mask); // Specify set of tags`
  - `mfc_read_tag_status_any(); // Wait for any DMA within group(s)`
  - `mfc_read_tag_status_all(); // Wait for all DMA within group(s)`
- There are many many different commands, check docs.

Simple DMA

- Effective Address (64-bit main memory address)
- Local Store Address (32-bit)
- Length (bytes)
- Tag (0-31)

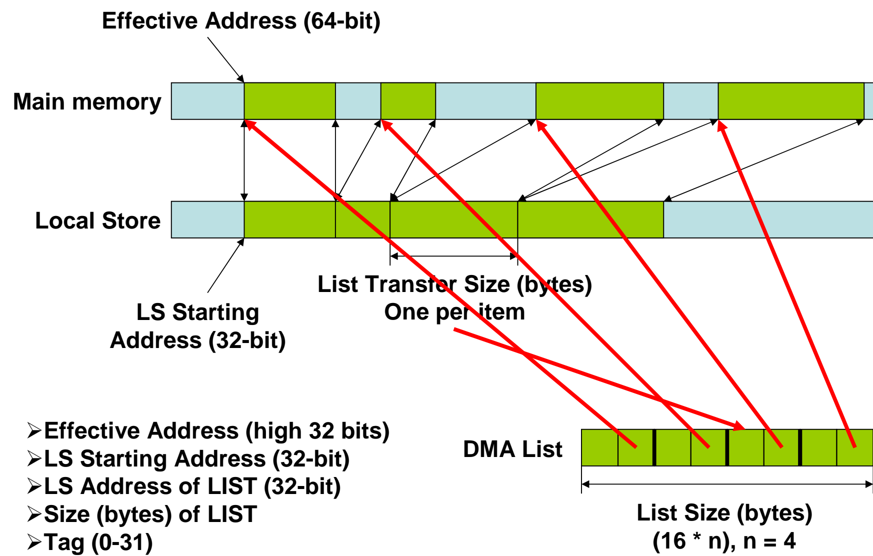
Limitations

- Maximum of 16KB per transfer.
- Length must be 1, 2, 4, 8, or n*16 bytes
- For 1, 2, 4, 8 bytes:
  - Source (and destination) must be naturally aligned (address divisible by length).
  - Source and destination addresses must have the same quadword offset (the four least significant bits are equal).
- For n*16 bytes:
  - Source and destination must be quadword aligned.

Simple DMA Example

```
// Get (from main memory to local store)
mfc_get(ptr, ea, size, tag, 0, 0);
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_any();


// Put (from local store to main memory)
mfc_put(ptr, ea, size, tag, 0, 0);
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_any();
```

DMA List



Limitations

- Maximum 16KB per list item.
- Maximum 2048 list items.
- List specification must be aligned on 8-byte boundary.
- Local Store address of each item is automatically aligned to quadword boundary.

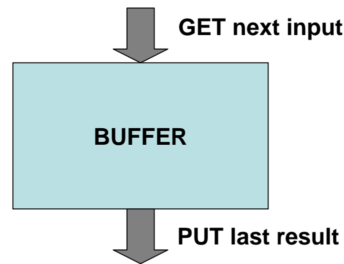
DMA List Example

```
mfc_list_element_t list[16] __attribute__((aligned(8)));

// Transfer possibly more than 16 KB with one transfer
void large_transfer( void *LS, unsigned long long EA, unsigned int nbytes ) {
   unsigned int i = 0;
   unsigned int tagid = 0;
   unsigned int listsize;
   unsigned int sz;
   unsigned int ealow = mfc_ea2l(EA);

   while( nbytes > 0 ) {
      sz = (nbytes < 16384) ? Nbytes : 16384;
      list[i].size = sz;
      list[i].eal  = ealow;
      nbytes -= sz;
      ealow  += sz;
      i++;
   }

   listsize = i * sizeof(mfc_list_item_t);
   mfc_getl(dst, EA, list, listsize, tagid, 0, 0);
   mfc_write_tag_mask(1 << tagid);
   mfc_read_tag_status_any();
}
```

Memory Order

- DMA Transfers are not ordered.
- Memory ordering (when needed) must be enforced by programmer.
- GET and PUT commands have two variants each:
  - GETF, PUTF
    - GET or PUT with FENCE
      - "This command is locally ordered with respect to all **previously** issued commands within the **same tag** group and command queue."
  - GETB, PUTB
    - GET or PUT with BARRIER
      - "This command and **all subsequent commands with the same tag** and command queue are locally ordered with respect to all previously issued commands within the same tag group and command queue."

FENCE Example



FENCED GET of next input ensures the PUT of the last result is finished.

SPE to SPE communication

- SPEs can communicate via DMA without leaving the EIB
- Looks identical to main memory DMA but the EA refers to special addresses which map to local stores.
- General procedure:
  - PPE creates SPE contexts
  - PPE maps local stores into PPE address space
  - PPE communicates the EA of the local stores to the SPEs
  - SPEs uses DMA to surgically put data in other SPEs local stores or get data from other SPEs local stores.
  - Synchronization is important.

SPE to SPE communication

- Mapping a local store into PPE address space:
  - `void *spe_ls_area_get(spe_ctxt);`
- Communicate the 64-bit EA (the void*) to an SPE.
- SPE must know the Local Store offset to calculate remote address.
  - Special case: if all SPEs run the same image, offset information of static data is local.

Static Analysis

Assembler and the `spu_timing` tool

Static Analysis

- Since the SPU and LS are predictable when all data is local (no MFC interaction) it is possible to get accurate information on pipeline states from assembler code.
- The SDK provides a tool – `spu_timinig` – to annotate assembler code with pipeline state information.
- Very useful to understand performance issues.
- We will look at how to use `spu_timing` to inspect the performance of the `vmul` example implementations.

Using the `spu_timing` tool

```
# Produce assembler code (vmul.s)
spu-gcc -O3 -S vmul.c

# Annotate assembler code (vmul.s.timing)
/opt/cell/sdk/usr/bin/spu_timing -running-count vmul.s
```

Brief Assembler Tutorial

- Labels (targets of branches):
  - `.L4`
- Registers
  - `$14`
- Branches
  - `brnz rt, lbl` // Branch to lbl if rt == 0
- Loads, stores
  - `lqx rt, ra, rb` // rt = *(ra + rb) Load with offset
  - `stqx rc, ra, rb` // *(ra + rb) = rc Store with offset
- Floating Point
  - `fm rt, ra, rb` // rt = ra * rb
  - `fma rt, ra, rb, rc` // rt = ra * rb + rc (Fused Multiply Add)
- Fixed Point
  - `a rt, ra, rb` // rt = ra + rb SIMD int add
  - `ai rt, ra, s10` // rt = ra + s10 SIMD int add constant (immediate)
- Data Movement
  - `cwx rt, ra, rb` // Make pattern (for shufb) to insert word into EA ra+rb
  - `rotqby rt, ra, rb` // Rotate ra left by rb BYTES and store in rt
  - `shufb rt, ra, rb, rc` // rt = spu_shuffle(ra, rb, rc) Shuffle bytes
- Comparison
  - `cgt rt, ra, rb` // rt = (ra > rb) SIMD int compare greater than

`spu_timing` Output Format

```
000000 0D 01                                     cgti  $2,$3,0
000000 1D 0123                                   shlqbyi  $12,$4,0
000001 0D  12                                    ori   $11,$5,0
000001 1D  1234                                  shlqbyi  $10,$6,0
000002 0D   2                                    nop   127
000002 1D   2345                                 biz   $2,$lr
000003 0D    34                                  ori   $9,$3,0
000003 1D    345678901234567                     hbrr  .L8,.L4
000004 0D     45                                 il $8,0
000004 1D     4                                  lnop
                                               .L4:
000006 0D      -67                               a  $17,$8,$12
000006 1D       678901                           lqx   $16,$8,$12
000007 0D        78                              a  $6,$8,$11
000007 1D        789012                          lqx   $15,$8,$11
000008 0D         89                             ai $9,$9,-1
000008 1D         890123                         lqx   $13,$8,$10
000009 1           9012                          cwx   $7,$8,$10
000012 1            --2345                       rotqby   $14,$16,$17
000013 1               3456                      rotqby   $3,$15,$6
000017 0                ---789012                fm $5,$14,$3
000023 1                    -----3456            shufb $4,$5,$13,$7
000027 1                          ---789012      stqx  $4,$8,$10
000028 0D                             89         ai $8,$8,4
                                               .L8:
000028 1D                             8901       brnz  $9,.L4
000029 1                               9012      bi $lr
```

Cycle counter | Pipeline and Dual issue | Instruction timing details | Assembler code

Inspecting vmul0 (slowest variant)

```
000000 0D 01                                cgti  $2,$3,0
000000 1D 0123                              shlqbyi  $12,$4,0
000001 0D  12                               ori   $11,$5,0
000001 1D  1234                             shlqbyi  $10,$6,0
000002 0D   2                               nop   127
000002 1D   2345                            biz   $2,$lr
000003 0D    34                             ori   $9,$3,0
000003 1D    345678901234567                hbrr  .L8,.L4
000004 0D     45                            il $8,0
000004 1D     4                             lnop
                                          .L4:
000006 0D      -67                          a  $17,$8,$12
000006 1D       678901                      lqx   $16,$8,$12
000007 0D        78                         a  $6,$8,$11
000007 1D        789012                     lqx   $15,$8,$11
000008 0D         89                        ai $9,$9,-1
000008 1D         890123                    lqx   $13,$8,$10
000009 1          9012                      cwx   $7,$8,$10
000012 1           --2345                   rotqby   $14,$16,$17
000013 1              3456                  rotqby   $3,$15,$6
000017 0               ---789012            fm $5,$14,$3
000023 1                   -----3456        shufb $4,$5,$13,$7
000027 1                         ---789012  stqx  $4,$8,$10
000028 0D                           89      ai $8,$8,4
                                          .L8:
000028 1D                           8901    brnz  $9,.L4
000029 1                             9012   bi $lr
```

Inspecting vmul4 (SW pipelining)

```
000427 1          -7890                     brz   $2,.L43
000428 0            89                      il $8,0
000429 0D            90                     ai $4,$12,-2
000429 1D            9                      lnop
                                          .L45:
000430 0D              01                   a  $14,$11,$8
000430 1D              012345               stqx  $9,$8,$6
000431 0D               12                  a  $13,$5,$8
000431 1D               1                   lnop
000432 0D                234567             fm $9,$10,$7
000432 1D                234567             lqd   $10,32($14)
000433 0D                 34                ai $4,$4,-1
000433 1D                 345678            lqd   $7,32($13)
000434 0d                  45               ai $8,$8,16
                                          .L48:
000435 1d                  -5678            brnz  $4,.L45
                                          .L43:
000439 0                     ---901234      fm $11,$10,$7
000440 0                         0123       shli  $15,$12,4
000444 0d                         ---45     a  $5,$6,$15
000446 1d 01                        --6789  stqd  $11,-16($5)
000447 1  012                          789  stqd  $9,-32($5)
000448 1  01                           89   bi $lr
```

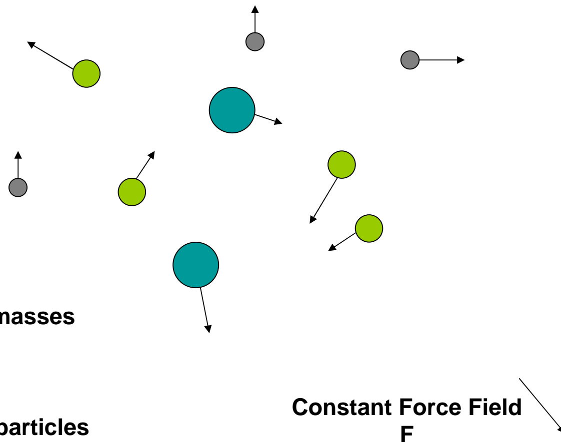
Inspecting vmul6 (fastest variant)

```
000341 0D                         12            ori   $4,$9,0
000341 1D 012345                  123456789     hbrr  .L34,.L30
000342 0D                          23           ori   $3,$6,0
000342 1D                          2345         fsmbi $9,0
                                              .L30:
000346 0D                           ---67       ai $9,$9,2
000346 1D 01                           6789     stqd  $8,0($3)
000347 0D 012                           789     fm $8,$13,$11
000347 1D 012                           789     stqd  $7,16($3)
000348 0D 0123                           89     fm $7,$12,$10
000348 1D 0123                           89     lqd   $13,64($4)
000349 0D 0                               9     cgt   $16,$14,$9
000349 1D 01234                           9     lqd   $11,64($5)
000350 0D 01                                    ai $3,$3,32
000350 1D 012345                                lqd   $12,80($4)
000351 0D  12                                   ai $4,$4,32
000351 1D  123456                               lqd   $10,80($5)
000352 0D   23                                  ai $5,$5,32
                                              .L34:
000352 1D   2345                                brnz  $16,.L30
                                              .L28:
000353 1      345678                            hbr   .L33,$lr
000355 0       -567890                          fm $9,$13,$11
000357 0         -789012                        fm $13,$12,$10
000358 0          8901                          shli  $12,$15,4
```

AoS || SoA

Storage Formats and Their Implications for SIMD'zation

Example: Euler Particle Simulation



- Particles with different masses
- 3D Position (pos)
- 3D Velocity (vel)
- 1/mass (inv_mass)
- Constant force field on particles
- Forward Euler scheme:
  - Time step (dt)
  - vel = vel + F * inv_mass * dt
  - pos = pos + vel * dt

Constant Force Field
F

Basic Code

```
typedef struct {
   float x, y, z, w;
} vec4D;

typedef struct {
   vec4D pos;
   vec4D vel;
   float inv_mass;
} Particle;

Particle p[PARTICLES];
float dt;
vec4D F;

for( time = 0; time < END_TIME; time += dt ) {
   for( i = 0; i < PARTICLES; i++ ) {
      p[i].pos.x = p[i].pos.x + p[i].vel.x * dt;
      p[i].pos.y = p[i].pos.y + p[i].vel.y * dt;
      p[i].pos.z = p[i].pos.z + p[i].vel.z * dt;
      p[i].vel.x = p[i].vel.x + F.x * p[i].inv_mass * dt;
      p[i].vel.y = p[i].vel.y + F.y * p[i].inv_mass * dt;
      p[i].vel.z = p[i].vel.z + F.z * p[i].inv_mass * dt;
   }
}
```

Array of Structures (AoS)

- Representation of one particle
  - Define a struct for the vector quantities (`pos, vel, F`)
  - Define a struct for the particle quantities (`pos, vel, inv_mass`)
- Representation of a collection of particles
  - Array of structs (AoS) defined above
- Positive:
  - Natural representation
  - Good encapsulation
- Negative:
  - Cumbersome to SIMD'ize
  - Suboptimal performance of SIMD'ized code

SIMD'zation of AoS

```
vector float *pos_v, *vel_v, *F_v;
vector float dt_v, F_inv_mass_v;
vector unsigned int pat = (vector unsigned int) {0xFFFFFFFF,
                                                 0xFFFFFFFF,
                                                 0xFFFFFFFF,
                                                 0x00000000};

F_v = (vector float *) &F;
dt_v = spu_and(spu_splats(dt), pat);

for( time = 0; time < END_TIME; time += dt ) {
   for( i = 0; i < PARTICLES; i++ ) {
      F_inv_mass_v = spu_mul(spu_and(spu_splats(p[i].inv_mass), pat), *F_v);
      pos_v  = (vector float *) &p[i].pos;
      vel_v  = (vector float *) &p[i].vel;
      *pos_v = spu_madd(*vel_v, dt_v, *pos_v);
      *vel_v = spu_madd(F_inv_mass_v, dt_v, *vel_v);
   }
}
```

Structure of Arrays (SoA)

- Different approach starts with the collection
- Representation of a collection of particles
  - One array per scalar component
    - `pos_x, pos_y, pos_z`
    - `vel_x, vel_y, vel_z`
    - `inv_mass`
  - One array index corresponds to one logical particle

- Positive:
  - Easy to SIMD'ize via loop unrolling (x4)
  - Good performance

- Negative:
  - Unnatural representation
  - Poor encapsulation
  - Input/output format often difficult to change

SIMD'zation of SoA

- SIMD'zation of SoA is simple since parallellization is across iterations instead of within an iteration as in AoS.
- Code can be made visually similar to scalar code but compute on four iterations simultaneously.
- Loop unrolling (x4) of the particle-loop (inner) is the key.

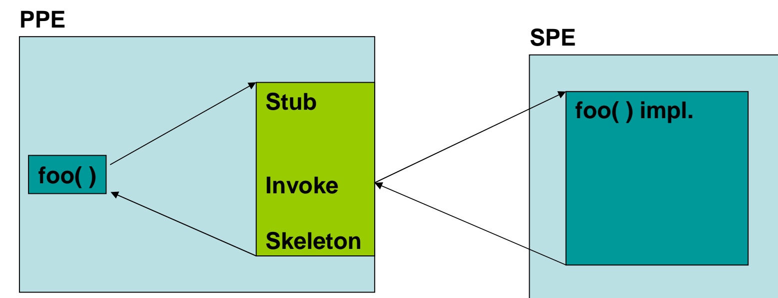
- Details left for you to fill in.

Programming Models

Function Offload
Computation-Acceleration
Streaming
Shared Memory

Function Offload

- Functions are moved to SPE.
- PPE interface oblivious of where code executes.
- In effect, an RPC implementation.



Computation-Acceleration

- PPE uses SPEs to accelerate compute-intensive portions of the code.
- Example: Video Editing
  - PPE handles
    - Control logic
    - GUI
    - Input/output
  - SPE handles
    - Encoding
    - Decoding
    - Transcoding

Streaming

- Data is "**streamed through**" an SPE.
  - Typically using multibuffering.
  - Kernel operates on data as it is streamed.
- Common programming model on GPUs
- Graphics codes often match this model well.
- Example: Audio Decoding
  - Input stream: Compressed audio
  - Kernel: Decode
  - Output stream: PCM audio

Shared Memory

- Threads on PPE and SPEs communicate via main memory.
- Mutexes, condition variables, semaphores, etc. can be implemented on the CELL.
- Difficult to take advantage of high bandwidth inter-SPE communication.

Further Information

- Official IBM Documentation library
  - http://www.ibm.com/developerworks/power/cell/documents.html
- C/C++ Language Extensions for Cell Broadband Engine Architecture
  - Here you will find information about the intrinsic functions
- SPE Runtime Management Library 2.2
  - How to manage SPEs from the PPE (load programs, open images, run contexts)
- Cell Broadband Engine Programmer's Tutorial (Handbook, Guide)
  - A more descriptive source (see also Handbook and Guide)
- Cell Broadband Engine Architecture (click on Hardware)
  - Answers your hardware related questions
- Google
  - Beware that the Cell programming APIs change rapidly and information you find on google is often stale and/or incorrect.
- Linux on your PS3
  - If you have access to a PS3, check out
    - http://www.ibm.com/developerworks/power/library/pa-linuxps3-1/