Cell Broadband Engine
Introduction and Communication

Lars Karlsson

April 28, 2009

Resources

(1) IBM DeveloperWorks Cell BE Resources.
`http://www.ibm.com/developerworks/power/cell/`
In particular,

- Programmer's Guide
- Programmer's Tutorial
- Cell BE Architecture Programming Handbook
- C/C++ Language Extensions for Cell BE Architecture
- SPE Runtime Management Library

(2) Programming the Cell Broadband Engine Architecture: Examples and Best Practices.
`http://www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf`

Header Files and Libraries

- PPU
  - Header files:
    `ppu_intrinsics.h`
    `libspe2.h` for managing SPUs
    `pthread.h` for OS threads
  - Libraries:
    `-lpthread`
    `-lspe2`
- SPU
  - Header files:
    `spu_intrinsics.h`
    `spu_mfcio.h`

Cell BE Architecture

- 1 PPU (PowerPC Processing Unit).
- 8 SPUs (Synergistic Processing Unit).
- Different instruction set architectures (ISAs).
- Each SPU has a 256KB fast local storage (LS).
  - Instead of cache.
  - Communication to/from main memory via direct memory access (DMA).
  - DMA concurrent with computation: offloaded to memory flow controller (MFC).
- The PPU has a modern and complex architecture with out-of-order execution, branch prediction, caching, etc. The PPU runs the operating system software.
- The SPUs have a SIMD architecture with 128-bit registers and a less complex architecture.
  - In-order execution.
  - No (automatic) branch prediction.
  - Two pipelines, dual issue.

Communication

- PPU has several levels of cache.
- PPU, memory, SPUs connected by the element interconnect bus (EIB).
- EIB consists of several ring networks with high bandwidth.
- There are several mechanisms for communication, e.g.,
  - DMA – general memory access.
  - Mailboxes – 32-bit messages.
  - Signals – notifications.
  - Events – events external to the SPU.
- The PPU maps parts of an SPU into its address space and forwards the EA to other SPUs for use in DMA operations.

Compiling and Linking

- Compiling and linking an SPU binary:

  ```
  spu-gcc -o spu_bin spu.c
  ```

- Embedding an SPU binary in a PPU object file:

  ```
  ppu-embedspu speobjectname spu_bin spu_bin-embed.o
  ```

  Use ppu32-embedspu for 32-bit PPU binaries.

- Compiling and linking a PPU binary containing an embedded SPU binary:

  ```
  ppu-gcc -o ppu_bin ppu.c spu_bin-embed.o -lspe2 -lpthread
  ```

  Use ppu32-gcc for 32-bit PPU binaries.

Launching SPE Code

- Initial declarations:

```
spe_context_ptr_t spe_ctx; // SPE context
spe_stop_info_t stop_info; // Status information
uint32_t entry = SPE_DEFAULT_ENTRY; // Entry point
```

- Create SPE context:

```
spe_ctx = spe_context_create(0, NULL);
```

- Load SPE object into the SPE context:

```
spe_program_load(spe_ctx, &spu_main);
```

- Run the SPE context until completion:

```
spe_context_run(spe_ctx, &entry, 0, NULL, NULL, &stop_info);
```

- Destroy context:

```
spe_context_destroy(spe_ctx);
```

Addresses and 32/64-bit Issues

- The PPU binaries use either 32-bit or 64-bit addresses (depending on the compilation).
- The SPU binaries always use 32-bit addresses.
- A pointer on the SPU can be stored in a 32-bit unsigned integer, `uint32_t` (see `<stdint.h>`).
- A pointer to main memory can always be stored in a `uint64_t`.
- The functions `mfc_ea2l(ea64)` and `mfc_ea2h(ea64)` extract the low and high 32 bits of a 64-bit address.

SPU `main`

- Declaration of SPU `main`:

```
int main(int speid, uint64_t argp, uint64_t envp);
```

- `speid`
  Numerical identifier of the context.
- `argp`
  Main memory address of "arguments".
- `envp`
  Main memory address of "environment".
- Both `argp` and `envp` are specified as parameters to `spe_context_run`:

```
spe_context_run(spe_ctx, &entry, 0, argp, envp, &stop_info);
```

  Therefore, they are simply two arbitrary 64-bit integers passed from the PPU to an SPU.

SPE-Initiated DMA To/From Main Memory

- Initial declarations:

```
uint32_t tag; // Tag
uint64_t ea = ...; // Effective address in main memory
volatile char data[256] __attribute__((aligned(128)));
```

- Reserve tag ID:

```
tag_id = mfc_tag_reserve();
```

- Enqueue a DMA `GET` command:

```
mfc_get((void*) data, ea, sizeof(data), tag, 0, 0);
```

- Wait for completion:

```
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_all();
```

- Free tag ID:

```
mfc_tag_release(tag_id);
```

DMA Restrictions

- Size must be 1, 2, 4, 8, 16, or a multiple of 16 bytes.
- Requests of size 1, 2, 4, 8, and 16 bytes: naturally aligned.
- Requests of a multiple of 16 bytes: 16 byte aligned.
- Source and destination effective addresses must have same 16 byte offset (same four least significant bits).
- Best performance when 128 byte aligned.

Managing DMA Restrictions: Alignment

- The compiler can be instructed to align variables to specific boundaries.
- Requesting 16 byte alignment:

  ```
  char data[256] __attribute__((aligned(16)));
  ```

  Use for
  - variables
  - struct members
- Padding the size of a *struct*:

  ```
  struct foo {
     char a;
  } __attribute__((aligned(16)));
  // sizeof(struct foo) == 16
  ```

- Setting a byte to 1, regardless of alignment:

  ```
  const char one[16] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
  uint32_t offset = (uint32_t) (ea & 0xf);
  mfc_put((void*) &one[offset], ea, 1, tag, 0, 0);
  ```

Managing DMA Restrictions: Odd Sizes and Misalignment

- Odd sized and/or misaligned requests must be reformulated as a composition of correctly sized requests.
- Code for finding the next larger multiple of 16:

```
#define ceil16(x) (((x) + 15) & ~15)
```

- How to PUT a string of any length:

```
char str[256] __attribute__((aligned(16))) = "Hello Cell!";
mfc_put((void*) str, ..., ceil16(strlen(str)), ...);
```

DMA Lists

- DMA lists are sequences of DMA commands that reads/writes segments from/to main memory.
- Segments in main memory can be discontinuous.
- The same restrictions as for basic DMA applies to DMA lists as well.
- Each local segment is automatically aligned to a 16 byte boundary.
- List element `struct`:

```
typedef struct {
   uint64_t notify   : 1; // stall-and-notify flag
   uint64_t reserved : 16;
   uint64_t size     : 15; // size in bytes
   uint64_t eal      : 32; // lower 32-bits of main memory address
} mfc_list_element_t;
```

Mailboxes: SPU Side

Each SPU has an incoming (4 `uint32_t`) and an outgoing mailbox (1 `uint32_t`).

SPU mailbox functions:

- Read next element in inbound mailbox (stalls if empty).
  `uint32_t spu_read_in_mbox(void)`
- Query the number of waiting elements in inbound mailbox.
  `uint32_t spu_stat_in_mbox(void)`
- Write element to outbound mailbox (stalls if full).
  `void spu_write_out_mbox(uint32_t data)`
- Query the available capacity of the outbound mailbox.
  `uint32_t spu_stat_ou_mbox(void)`

Mailboxes: PPU Side

- Read one or more elements from the SPU (nonblocking). Returns the number of elements actually read.

```
int spe_out_mbox_read(spe_context_ptr_t spe, // SPE context
                      unsigned int *mbox_data, // buffer
                      int count) // number of elements
```

- Query the number of available outgoing elements.

```
int spe_out_mbox_status(spe_context_ptr_t spe)
```

- Write one or more elements to the SPU. Returns the number of elements actually written.

```
int spe_in_mbox_write(spe_context_ptr_t spe, // SPE context
                      unsigned int *mbox_data, // buffer
                      int count, // number of elements
                      unsigned int behavior)
```

- Query the available capacity on the incoming mailbox.

```
int spe_in_mbox_status(spe_context_ptr_t spe)
```

Double Buffering

```
i = 0;
// Load buffer 0.
mfc_get(buf[i], ..., tag[i], ...);
while( ! done ) {
   // Load next buffer (barrier ensures store has finished).
   mfc_getb(buf[i^1], ..., tag[i^1]...);
   // Wait for buffer i to finish loading.
   wait(tag[i]);
   // Compute on buffer i.
   // Store buffer i.
   mfc_put(buf[i], ..., tag[i], ...);
   // Switch buffers.
   i ^= 1;
}
// Wait for last buffer to finish loading.
wait(tag[i]);
// Compute on last buffer.
// Store last buffer.
mfc_put(buf[i], ..., tag[i], ...);
// Wait for last store.
wait(tag[i]);
```

Atomic Operations

- Atomic operations can be implemented using the general concepts of locked lines and reservations.
- Structure of atomic operation:
  1. Load the variable and add reservation.
  2. Modify the variable.
  3. Store the variable if the reservation was not lost.
- The first and third steps are supported by atomic DMA commands.
- The second step is application specific which makes the whole construction very general.

Atomic Operations: Implementation

- Load with reservation:

```
mfc_getllar((void*) ls, ea, 0, 0); // 128B and 128B aligned
mfc_read_atomic_status();
```

- Store if reserved (status != 0 means reservation lost):

```
mfc_putllc((void*) ls, ea, 0, 0);
status = mfc_read_atomic_status() & MFC_PUTLLC_STATUS;
```

- Store regardless of reservation:

```
mfc_putlluc((void*) ls, ea, 0, 0);
mfc_read_atomic_status();
```

Atomic Operations: Skeleton

```
uint32_t status;
// Loop until successful.
do {
   // Load 128B variable and add reservation.
   mfc_getllar((void*) ls, ea, 0, 0);
   // Wait for completion.
   mfc_read_atomic_status();
   // Update variable...
   // Try to store the variable.
   mfc_putllc((void*) ls, ea, 0, 0);
   // Wait for completion and check status.
   status = mfc_read_atomic_status() & MFC_PUTLLC_STATUS;
} while( status );
```

High Resolution Timings

- The SPU decrementer can be used as a high resolution timer.
- ```
  spu_write_decrementer(0x7fffffff); // initialize decrementer
  uint32_t t1 = spu_read_decrementer(); // read initial value
  // do work...
  uint32_t t2 = spu_read_decrementer(); // read final value
  uint32_t diff = t1 - t2; // elapsed time (in ticks)
  ```
- Conversion to seconds:

  ```
  float timebase = 266664960.f; // Machine dependent (ticks / sec).
  float sec = diff / timebase;
  ```
  ```