Cell programming mini-reference

Cell documentation

Most of the documentation for Cell is at http://www-128.ibm.com/developerworks/power/cell/. Click on the Docs tab, then click on Documentation archive. When browsing through the PDF documentation, Acrobat's Bookmarks tab contains the table of contents. For library references, this contains a listing of all functions.

SPE runtime management library (currently the 4th entry in the list) contains information on the SPE management functions (`libspe`) available to the PPU.

PPU & SPU C/C++ language extension specification (5th entry) contains information on the basic library available to SPU programs (DMA/mailbox access functions, vector data types and intrinsics).

SIMD math library specification contains information on SIMD math functions available to both SPU and PPU programs.

For information on other libraries available on the SPU, look at /opt/ibm/cell-sdk/prototype/docs/libraries_SDK.pdf on the PS3. The header files in /usr/spu/include/ may also be useful, especially for the SIMD math functions.

More detailed information about Cell is available in the *Cell Broadband Engine programming handbook* and *Cell Broadband Engine Architecture*.

SPE thread issues

SPE threads behave differently from "normal" threads. The OS currently runs SPE threads on physical SPEs in FIFO order (with some exceptions): SPE threads are not pre-empted, and if more SPE threads are created than there are physical SPEs, the extra threads will wait until the running threads have exited (i.e., returned from `main`) before they begin executing. This applies across SPE threads from all processes: if one process is using all SPEs, no other process can use the SPEs. When an SPU program stalls due to a mailbox or signal notification register access or DMA completion wait, it still occupies its SPE.

Generally, this means that you shouldn't create more threads than there are SPEs. Only 6 SPEs are enabled on the PS3s.

DMA issues

(A word on Cell is 4 bytes; the documentation often refers to quadwords, which are 16 bytes.)

Alignment

DMA transfers must be a multiple of 16 bytes (up to a maximum of 16 KB) and *aligned* on a 16 byte boundary (i.e., the local store and memory addresses the DMA starts at must both be divisible by 16).

Alternatively, Cell also supports DMA transfers of 1, 2, 4, or 8 bytes that are *naturally aligned* (the starting local store and memory addresses must be divisible by the size of the transfer) and have the same local store and memory address offsets with a 16-byte block (e.g., local store address mod 16 = memory address mod 16).

Transfers of 16 bytes or less are *atomic*: all processors observe the transfer as occuring in a single operation. For larger transfers, processors may be able to observe that e.g. the first 16 bytes of the transfer have occured but the rest has not.

Structure padding

Use the `aligned` attribute:

```
__attribute__((aligned(n)))
```

to pad and align structure declarations to a multiple of *n* bytes (must be a power of 2, within reasonable limits).

```
struct S {
  int a, b, c;
} __attribute__((aligned(16)));
```

This pads the size of the structure to a multiple of 16 bytes (`sizeof(struct S)` becomes 16 instead of 12), and ensures that all variables of this type are 16-byte aligned.

Dynamic allocation

`malloc` on the SPEs allocates memory that is 8-byte aligned. To allocate memory with larger alignments (on both the SPU and PPU), use `malloc_align`:

```
void *malloc_align(size_t size, unsigned int n)
```

which allocates memory that is 2^n-byte aligned (free this memory using `free_align`). To use `malloc_align/free_align`, you must `#include <libmisc.h>` and link with `libmisc` (add `-lmisc` to the `IMPORTS` variable in your SPU program makefile).

SPE management functions

This section describes some useful SPE management functions. These functions are part of `libspe` and are available only to the PPU.

Full documentation for `libspe` is in the *SPE runtime management library* document.

Thread management

```
spe_create_thread(gid, program, argp, envp, -1, flags)
```

Creates (and starts, if a physical SPE is available) an SPE thread running `program`.

- `gid` specifies the thread group (see spe_create_group); specify 0 if you're not interested (this creates a new thread group)
- `argp` and `envp` are passed to the SPU program's `main` function
- `flags` can be 0, or `SPE_MAP_PS` if other SPEs need to access the mailboxes/signals of this program. There are also other flags (see documentation).

Returns the thread ID (an opaque value).

```
spe_wait(speid, NULL, options)
```

Waits or polls for SPE thread `speid` to exit.

- `options` can be 0 to wait or `WNOHANG` to poll

```
spe_get_ls(speid)
```

Returns the address of SPE thread `speid`'s local store. The PPU or other SPUs can use this address to access this SPE's local store.

```
spe_get_ps_area
```

Returns the address of an SPE thread's mailboxes and signal notification registers (see documentation). Other SPUs can use this address to write to this SPE's mailboxes/signals.

Mailboxes

```
spe_write_in_mbox(speid, data)
```

Writes `data` to SPE thread `speid`'s inbound mailbox. Does not stall; overwrites last entry if no space is available.

```
spe_stat_in_mbox(speid)
```

Returns the space available in SPE thread `speid`'s inbound mailbox. This is the number of times `spe_write_in_mbox` can be safely called without overwriting an entry.

```
spe_read_out_mbox(speid)
```

Reads from SPE thread `speid`'s outbound mailbox. Does not stall; returns -1 if no entry is present.

```
spe_stat_out_mbox(speid)
```

Returns the number of entries present in SPE thread `speid`'s mailbox. This is the number of times `spe_read_out_mbox` can be safely called. This is always 0 or 1, since the outbound mailbox has capacity for only one entry.

DMA

```
spe_mfc_get
spe_mfc_put
spe_mfc_read_tag_status
```

Analogous to SPU's DMA functions (see documentation).

SPU functions

This section describes some useful DMA and communication functions for SPU programs. Full documentation is in section 4 of the *PPU & SPU C/C++ language extension specification*.

DMA

```
mfc_get(dest_lsa, src_addr, num_bytes, tag, 0, 0)
mfc_put(src_lsa, dest_addr, num_bytes, tag, 0, 0)
```

Starts a DMA transfer.

- `tag` specifies a 5-bit (0 - 31) tag for the transfer

DMA completion

```
mfc_write_tag_mask(mask)
```

Writes a mask specifying the DMA tags subsequent `mfc_read_tag_status*` calls wait for.

```
mfc_read_tag_status_all()
```

Waits until DMA transfers using every tag in the last specified tag mask have completed.

`mfc_read_tag_status_any()`

Waits until any tag in the last specified tag mask has completed.

`mfc_read_tag_status_immediate()`

Returns a mask indicating the tags in the last specified tag mask that have completed. Does not stall.

Mailboxes

`spu_read_in_mbox()`

Reads from the inbound mailbox. Stalls until an entry is present.

`spu_stat_in_mbox()`

Returns the number of entries present in the inbound mailbox. This is the number of times `spu_stat_in_mbox` can be safely called without stalling.

`spu_write_out_mbox(data)`

Writes `data` to the outbound mailbox. Stalls until space is available.

`spu_stat_out_mbox()`

Returns the space available in the outbound mailbox. This is the number of times `spu_write_out_mbox` can be safely called without stalling. This is always 0 or 1, since the outbound mailbox has capacity for only one entry.