Introduction to Cell/B.E. Architecture
IBM PowerXcell 8i

Jordi Caubet. IBM Spain.
IBM Innovation Initiative @ BSC-CNS

BSC-PRACE code porting and optimization Workshop,
Barcelona, October, 21st-23rd, 2009
CBEA Motivation

- Partnership of Sony, Toshiba, IBM (STI) created in 2001
  - Joint STI Design Center established in Austin, Texas, USA
  - US $400M joint investment

- Focus on developing a technology to:
  - Significant increases in computational performance
  - Good power efficiency
  - At low cost

- Attack on the “Power Wall”
  - Non Homogenous Coherent Multiprocessor
  - High design frequency @ a low operating voltage with advanced power management

- Attacks on the “Memory Wall”
  - Streaming DMA architecture
  - 3-level Memory Model: Main Storage, Local Storage, Register Files

- Attacks on the “Frequency Wall”
  - Highly optimized implementation
  - Large shared register files and software controlled branching to allow deeper pipelines
Traditional SMP architecture
Cell Broadband Engine Architecture

[Diagram showing the architecture of the Cell Broadband Engine]

- PPE (Processing Power Element)
- Thread
- SPE (Synergistic Processing Element)
- Interconnect
- Memory Interface
- I/O Interface
PowerPC Processor Element (PPE)

- Implements standard PowerPC instruction set architecture (ISA)
  - Uses Linux ppc64 port
  - Optimised for control intensive code
- Can directly access all system memory via a 2-level cache hierarchy
- Superscalar core
- Simultaneous multi-threading
  - Looks like 2-way SMP to OS
- VMX SIMD unit
  - 128 bit SIMD registers
Synergistic Processor Elements (SPE)

- **Implements variation of PowerPC ISA**
  - Optimised for compute intensive code using SIMD
  - Lots of SIMD registers (128 x 128 bit)

- **Can only directly address local store (LS)**
  - Only 256KB per SPE

- **Access to system memory requires DMA to/from LS**
  - Memory access operations operate asynchronously and in parallel with computation
  - LS is effectively a software managed “cache”
  - Pre-fetching can more effectively hide memory latency
  - 16 concurrent DMAs per SPE

- **DMA also used to access LS in other SPEs and I/O**
Local Store

- 256 KB addressable local memory available to SPU Core.
  - Text code, data, heap and stack.
  - Never misses
    - No tags, backing store, or prefetch engine
    - Predictable real-time behavior
    - Less wasted bandwidth
    - Easier programming model to achieve very high performance
    - Software managed caching
    - Large register file –or– local memory
    - DMA’s are fast to setup – almost like normal load instructions
    - Can move data from one local store to another

- No translation
  - Multiuser operating system is running on control processor
  - Can be mapped as system memory - cached copies are non-coherent wrt SPU loads/stores

- DMA commands move data between system memory & Local Storage
- DMA commands are processed in parallel with software execution
  - Double buffering
  - Software Multithreading
SPE Considerations

- SPU Core is a native SIMD architecture (single-instruction multiple data)
  - Scalar code not very efficient. SIMDimze your code (see MatMult handson exercise).
  - 128 registers 128-bit wide.

```c
float *A, *B, *C;
for (ii=0; ii<N; ii++)
{   
    A[ ii ] = B[ ii ] + C[ii];
}
```

- SPU Core is in order dual issue core and dual issue requires alignment to instruction type.
  - Dual issue occurs in a fetch group when first instruction goes to the even pipeline and second to the odd

```c
vector float *A, *B, *C; (*) Assumes that N is divisible by 4
for (ii=0; ii< (N>>2); ii++)
{   
    A[ ii ] = spu_add( B[ ii ], C[ii] );
}
```

- No branch prediction and branch miss-predicts cost is high.
  - Avoid branches by:
    - using inline code
    - compute both paths and use select instruction
    - unroll loops
    - use branch hint instruction
Element Interconnect Bus (EIB)

- Bus data ring for internal communication
  - Four 16 byte data rings, supporting multiple transfers
  - 96B/cycle peak bandwidth. Over 100 outstanding requests

- Each EIB Bus data port supports 25.6GBytes/sec* in each direction
  - Four 16B data rings connecting 12 bus elements (2 clockwise / 2 counter-clockwise)

- The EIB Command Bus streams commands fast enough to support:
  - 102.4 GB/sec for coherent commands
  - 204.8 GB/sec for non-coherent commands.

* The above numbers assume a 3.2GHz core frequency – internal bandwidth scales with core frequency.
CBEA Implementations – Chips

- **Cell Broadband Engine**
  - First disclosed in 2005
  - Used in:
    - IBM QS20/QS21 blades
    - Sony PS3
  - 1 PPE + 8 SPEs
  - RAMBUS memory
    - Up to 1GB per chip
  - Optimised for **single** precision floating point
    - Peak SP: 204.8 Gflops
    - Peak DP: 12.8 Gflops

- **IBM PowerXcell 8i**
  - Introduced in 2008
  - Used in:
    - IBM QS22 blades
  - 1 PPE + 8 SPEs
  - DDR2 memory
    - Up to 16GB per chip
  - Optimised for **double** precision floating point
    - Peak SP: 204.8 Gflops
    - Peak DP: **102.4 Gflops**
CBEA Implementations – IBM QS22

- Introduced in 2008
- Optimized for DP floating point applications
  - First platform to use IBM PowerXcell 8i
  - Up to 32 GB of memory
  - NUMA design
  - No internal disk. Optional small flash disk for OS installation.
  - Peak SP: 409.6 Gflops
  - Peak DP: 204.8 Gflops
Structuring Applications to Exploit CBEA

- **CBEA based systems can implement parallelism at multiple levels:**
  - SIMD units within the PPE and each SPE
  - PPE + Multiple SPEs within each CBEA processor
    - Share global memory, independent local stores
  - Multiple CBEA processors within each “node” (e.g. IBM QS22 blade)
    - Share memory, although latency & bandwidth varies
    - i.e. Non-Uniform Memory Access (NUMA)
  - Multiple “nodes” in a system (e.g. Maricel PRACE Prototype System)
    - Don’t share memory – must use message passing (MPI)

- **Many ways to decompose application function and data to exploit the available parallelism …**
Application Partitioning Across PPE and SPEs

- Several models are possible and are often classified as either:
  - PPE centric
    - Main application runs on PPE
    - PPE offloads work to SPEs
  - SPE centric
    - Main application(s) run on SPEs
    - PPE provides services to SPEs

- Most programmers choose a PPE centric model
PPE centric model programming approach

Cell BE Porting & Optimizing

- Port app to Linux, if needed
- Port app to Power, run on PPE
- Begin moving function to SPE’s
- Tune SIMD
- Optimizing function on SPE’s

- Exploit Parallelism at Task Level
- Exploit Parallelism at instruction / data level
- Data and Instruction Locality Tuning

Writing for Cell BE speeds up code on all multi-core architectures because it uses the same parallel best practices – Cell architecture just gains more from them because of its design.
Asymmetric Thread Model

- Low-level model where programmer explicitly creates threads that run on PPE and SPEs
  - Many other models are ultimately built on top of this model
  - Application frameworks or compiler support for higher level models can hide thread creation and inter-thread communication from programmer

- Separate code is developed for PPE and SPEs
  - Exposes different instruction set architectures (ISAs)

- Fundamental model provided by the SDK's SPU Runtime Management Library.
Function Offload Model

- Main application runs on PPE
- Selected computationally intensive functions are offloaded to SPEs
- Main application uses glue code to invoke function on SPEs
  - Original function is replaced by some code that ends invoking the same functionality on the SPE.
- CellSs programming models support this model.
  - Compiler offload functions automatically based on user specified pragmas.
  - Data is managed by CellSs runtime system.
  - Runtime manages task execution and task dependencies automatically.
- ALF framework also supports the Function offload model
  - Handles data decomposition and scheduling of PPEs
Symmetric Multi-processor Model

- Programmer views combination of PPE and SPEs as a SMP i.e. sharing a single global memory
- **OpenMP API** used to express parallelism
- Compiler hides differences in architecture between PPEs and SPEs from programmer
- Compiler manages data movement to/from global memory and SPE local stores
- Still somewhat experimental …
CBEA Synergy

- **Cell is not a collection of different processors, but a synergistic whole**
  - Operation paradigms, data formats and semantics consistent
  - Share address translation and memory protection model

- **PPE** for
  - operating systems and program control

- **SPE**
  - optimized for efficient data processing
    - SPEs share Cell system functions provided by Power Architecture
    - MFC implements interface to memory
      - Copy in/copy out to local storage

- **EIB integrates system** as data transport hub

- **PowerPC provides system functions**
  - Virtualization
  - Address translation and protection
  - External exception handling
What is next ...

- IBM CBEA Software Development Kit (CBEA)
  - SPE Runtime Management Library

- CBEA Basic Communication Mechanisms
  - Memory Flow Controller
    - DMA transfers
    - Mailboxes
    - Signals
IBM CBEA Software Development Kit (SDK)

- **Product and free “developer” versions available**
  - Developer version runs under Fedora 9 and product version RHEL 5.1/5.2/5.3
  - Two download sources: *IBM developerWorks* and *Barcelona Supercomputing Center (BSC-CNS)*
    - Download from IBM developerWorks Portal includes:
      - Compilers (ppuxlc, ppuxlf, spuxlc, spuxlf, cbexlc)
      - Plug-in for Eclipse IDE (download base IDE from eclipse.org)
      - Libraries and frameworks
      - Full system simulator
      - Lots of documentation!
    - Download from BSC-CNS includes OSS components:
      - SPE Runtime Management Library
      - Linux kernel for Fedora 9
      - GNU Toolchain (ppu-gcc, spu-gcc)
- **Developer version supported only via developerWorks discussion forum**
IBM Cell SW Environment

STANDARDS – HW (CBEA) SW (ABI, Language, Assembly, SIMD math, libspe2)

Hardware – QS21, QS22, Soma CAB
IBM Full System Simulator

Enhanced Linux – RHEL 5.2/5.3
Fedora 9

Application Libraries
SIMD math, MASS/MASSV, crypto,
Monte Carlo RNG, FFT, BLAS, LAPACK

SPE Runtime Management Library (libspe2)
SPU system library (C99/posix, __ea cache, spu_timers)

Examples, Demos, Benchmarks

Security SDK

Prog Env
ALF, DaCS
Performance Tools
spu_timing, asmVis,
PDT/PDTR, FDPR-Pro,
Cell Perf Counter,
Oprofile, Code Analyzer

Performance Tools

Compilers
gnu C/C++, Fortran, Ada
XL – C/C++, Fortran,
single source compiler

IDE – Integrated Dev Env
Development Environment

VPA – Visual Perf Analyzer

PTP – Parallel Tools Platform

Runtime Environment

gdb – combined debugger

GNU binutils

Examples, Demos,
Benchmarks

Application Libraries

Security SDK

Prog Env
ALF, DaCS

Examples, Demos,
Benchmarks

Application Libraries
SIMD math, MASS/MASSV, crypto,
Monte Carlo RNG, FFT, BLAS, LAPACK

SPE Runtime Management Library (libspe2)
SPU system library (C99/posix, __ea cache, spu_timers)

Enhanced Linux – RHEL 5.2/5.3
Fedora 9

Hardware – QS21, QS22, Soma CAB
IBM Full System Simulator

STANDARDS – HW (CBEA) SW (ABI, Language, Assembly, SIMD math, libspe2)
SPE Runtime Management Library

- **Basic low-level API for accessing SPEs**
  - Supports the **Asymmetric Thread Model**
  - SPE thread programming model for Cell/B.E.

- **Provides functions to:**
  - Load code into SPE local stores from PPE
  - Initiate execution of code on SPEs
  - Handles SPEs as virtual objects: SPE Context
  - Access SPE to SPE and SPE to PPE IPC mechanisms

- **Separate source files for PPE and SPE code**
- **SPE code can be loaded from a library or appended to PPE executable**
SPE Runtime Management Library - Terminology

- **Main Thread**
  - Main application thread on PPE.

- **SPE Thread**
  - `pthread` created by the Main Thread on the PPE that manages and runs a SPE context.

- **SPE Context**
  - “Virtual SPE” that holds the persistent information.

- **Gang Context**
  - Groups a set of SPE context that should be treated as a gang.
SPE Runtime Management Library - Functions

- **Create / destroy SPE Contexts and Gang contexts**
  - spe_context_create, spe_context_destroy, spe_gang_context_create, spe_gang_context_destroy

- **Load SPE objects into SPE local store**
  - spe_image_open, spe_image_close, spe_program_load

- **Access the Memory Flow Controller (MFC)**
  - MFC proxy command issue
  - MFC proxy tag-group completion facility
  - Mailbox facility
  - SPE signal notification facility

- **Direct SPE access for applications**
  - spe_ls_area_get, spe_ls_size_get, spe_ps_area_get

- ...
Example: Single Thread

**PPE program**

```c
#include <errno.h>

#include <libspe2.h>
extern spe_program_handle_t hello_spu;

int main( void) {
    spe_context_ptr_t speid;       // Structure for an SPE context
    unsigned int flags = 0;  unsigned int entry = SPE_DEFAULT_ENTRY;
    void * argp = NULL;  void * envp = NULL;
    spe_stop_info_t stop_info;
    int rc;

    // Create an SPE context
    speid = spe_context_create(flags, NULL);
    if (speid == NULL)  { perror( ... ); return -2; }

    // Load an SPE executable object into the SPE context's local store
    if (spe_program_load(speid, &hello_spu) )
        { perror("... "); return -3; }

    // Run the SPE context
    rc = spe_context_run(speid, &entry, 0, argp, envp, &stop_info);
    if (rc < 0) perror( ... );

    spe_context_destroy(speid); // Destroy the SPE context
    return 0;
}
```

**SPE program hello_spu**

```c
#include <stdio.h>

int main( unsigned long long speid, unsigned long long argp, unsigned long long envp ) {
    printf("Hello World!\n");
    return 0;
}
```

- **Main Thread is a SPE Thread**
- **Basic scheme:**
  - Create an SPE context
  - Load an SPE executable object into the SPE context's local store
  - Run SPE context.
    - **Synchronous call.** Calling PPE thread blocks until SPE finishes.
    - It transfers control to the operating system requesting the actual scheduling of the context to a physical SPE in the system
  - Destroy SPE context.
Build Process
Example: Single Asynchronous Thread

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

// Structure for an SPE thread
typedef struct ppu_pthread_data{
    spe_context_ptr_t context;
    pthread_t pthread;
    unsigned int entry;
    unsigned int flags;
    void *argp;
    void *envp;
    spe_stop_info_t stopinfo;
} ppu_pthread_data_t;

// pthread function to run the SPE context
void *ppu_pthread_function(void *arg)
{
    ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
    int rc;
    rc = spe_context_run( datap->context, &datap->entry, 
        datap->flags, datap->argp, 
        datap->envp, &datap->stopinfo);
    pthread_exit(NULL);
}

extern spe_program_handle_t hello_spu;

int main(void)
{
    ppu_pthread_data_t data; // SPE Thread data
    data.context = spe_context_create(0, NULL);
    spe_program_load(data.context, &hello_spu);
    data.entry = SPE_DEFAULT_ENTRY;
    data.flags = 0;
    data.argp = NULL;
    data.envp = NULL;
    pthread_create( &data.pthread, NULL, &ppu_pthread_function, &data);
    pthread_join(data.pthread, NULL);
    spe_context_destroy(data.context);
    return 0;
}
```

- **Main Thread is NOT a SPE Thread**
- **Basic scheme:**
  - Create SPE context + Load SPE executable
  - Create a pthead (SPE thread)
    - Run SPE context in the pthead. **Synchronous call.** Calling thread blocks; not master pthread.
  - Join pthead and destroy SPE Context.
Example: Multi-threaded Asynchronous

- Create many SPE Threads (pthread) as concurrent SPE contexts are required

- Basic scheme:
  - *Create N SPE contexts*
  - Load the appropriate SPE executable object into each SPE context’s local store
  - Master thread *creates N SPE threads* (pthread_create)
    - In *each SPE Thread run one SPE Context*. Synchronous call.
    - Exit the SPE thread
  - Master Thread *joins all SPE Threads*
  - *Destroy all SPE Contexts*
CBEA Basic Communication Mechanisms

- **DMA transfers**
  - Used to move data and instructions between main storage and Local Storage (LS).
  - Asynchronous in order to hide memory latency and transfer overhead by moving in parallel with SPU computation.

- **Mailbox messages**
  - Used for control communication between an SPE, PPE and other devices.
  - Each SPE has two mailboxes for sending and one for receiving.
  - Mailboxes hold 32-bit messages.

- **Signal notification**
  - Used for control communication from PPE and other devices.
  - Uses 32-bit registers that can be configured for one-sender-to-one receiver or many-sender-to-one-receiver signalling.

- **All three are implemented and controlled by the SPE’s MFC**
Storage Domain Concepts

- An SPE program references its own Local Store using a **local store address (LSA)**.
  - LD/ST instructions can only access the SPE Local Store.
  - Transfers from main memory are explicit through DMA transfers.

- **Effective Address** is an address on main memory as seen by a normal PowerPC application.
  - DMA transfer requests contain both an LSA and an EA.

- **SPE Local Stores** are **mapped** on main memory and have an Effective Address (EA)
  - It allows SPEs to use DMA to directly transfer data between their LS to another SPE’s LS. Very efficient.

- Each **SPE MFC** is also mapped so PPE can communicate with SPEs using that memory
DMA Transfers

- MFC Commands that transfer data are referred as DMA Commands
- Transfer direction for DMA commands is referenced from the SPE:
  - From main memory to Local Store: GET
    
    (void) mfc_get( volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, 
    uint32_t tid, uint32_t rid )

  - From Local Store to main memory: PUT
    
    (void) mfc_put( volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, 
    uint32_t tid, uint32_t rid )

  ls: local-storage address
  ea: effective address in main storage
  size: DMA transfer size in bytes
  tag: DMA group tag id
  tid and rid

- 5-bit DMA Tag
  - Same identifier can be used for multiple commands.
  - Used to determine the status of an entire tag group or command or check/wait on the completion.
  - Tagging is useful when using barrier (mfc_getb, mfc_putb) and fence (mfc_getf, mfc_putf) commands.
DMA Characteristics

- DMA transfers
  - Transfer Sizes
    - 1, 2, 4, 8, 16 and multiple of 16 bytes
    - Maximum transfer size is 16 KB
    - Peak performance when size is multiple of 128 bytes.
  - Aligment
    - src and dest address must have the same 4 least significant bits.
    - Transfers < 16 bytes → naturally aligned based on transfer size
    - Transfers >= 16 bytes → aligned at 16-byte boundary.
    - 128-byte alignment for src and dest provides best performance.

- DMA command queues per SPU
  - PPE-initiated requests: 8-element queue (8 request for all SPEs)
  - SPU-initiated requests: 16-element queue (Always preferable. 16 requests for SPE)

- DMA lists
  - a single DMA command that can cause execution of a list of transfer requests (in LS)
  - lists implement scatter/gather operation
  - a list can contain up to 2K transfer requests
  - array of effective addresses and their sizes to get/put
  - see Hands-on2 for a DMA list example.
PPE initiated DMA request

- PPE can access SPE through the MMIO interface.
- Two options to initiate a DMA:
  - by MFC functions
    - Simple to use
    - Less development time
  - Direct problem state access (or Direct SPE access)
    - Offer better performance than MFC functions

SPE MFC Proxy Command Issue
- spe_mfcio_put, spe_mfcio_putb, spe_mfcio_putf
- spe_mfcio_get, spe_mfcio_getb, spe_mfcio_getf

SPE MFC Proxy Tag-Group Completion Facility
- spe_mfcio_tag_status_read
SPE initiated DMA request

From main memory to Local Store

```
#include <spu_mfcio.h>

inline void dma_memory_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();
}
```

Program a DMA GET commad from `mem_addr` in main memory to `ls_addr` in the local store.

Set tag mask to determine which tag ID to notify upon completion.

Wait until all tagged DMA commands completed.

From Local Store to main memory

```
#include <spu_mfcio.h>

inline void dma_ls_to_memory(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();
}
```

Program a DMA PUT commad from `ls_addr` in local store to `mem_addr` in main memory.
DMA Example: Get a string from main memory and modify it

**PPE program**

```c
#include <stdio.h>
#include <string.h>
#include <libspe2.h>

extern spe_program_handle_t getbuf_spu;

unsigned char buffer[128] __attribute__ ((aligned(128))); // local buffer

//spe context
spe_context_ptr_t speid;
unsigned int flags = 0;
unsigned int entry = SPE_DEFAULT_ENTRY;
spe_stop_info_t stop_info;

int main (void)
{
    strcpy (buffer, "Good morning!");
    printf("Original buffer is %s\n", buffer);
    speid = spe_context_create(flags, NULL);
    spe_program_load(speid, &getbuf_spu);
    rc = spe_context_run( speid, &entry, 0,
                          buffer);
    spe_context_destroy(speid);
    printf("New modified buffer is %s\n", buffer);
    return 0;
}
```

**SPE program**

```c
#include <stdio.h>
#include <string.h>
#include <spu_mfcio.h>

unsigned char buffer[128] __attribute__ ((aligned(128))); // local buffer

// spu program
extern spe_program_handle_t getbuf_spu;

// global buffer
unsigned char buffer[128] __attribute__ ((aligned(128)));

//spe context
spe_context_ptr_t speid;
unsigned int flags = 0;
unsigned int entry = SPE_DEFAULT_ENTRY;
spe_stop_info_t stop_info;

int main( unsigned long long speid, unsigned long long argp, unsigned long long envp )
{
    int tag = 31, tag_mask = 1<<tag;
    // DMA in buffer from PPE
    mfc_get( buffer, (unsigned long long) argp,
            128, tag, 0, 0);
    mfc_write_tag_mask(tag_mask);
    mfc_read_tag_status_any();
    printf("SPE received buffer "%s"
", buffer);
    // modify buffer
    strcpy (buffer, "Guten Morgen!");
    printf("SPE sent to PPU buffer "%s"
", buffer);
    // DMA out buffer to PPE
    mfc_put( buffer, (unsigned long long) argp,
             128, tag, 0, 0);
    mfc_write_tag_mask(tag_mask);
    mfc_read_tag_status_any();
    return 0;
}
```

Effective address of the global buffer passed as argument
Mailbox messages

- Used for **control communication between SPE, PPE and other devices**.
- To send small messages *up to 32 bits in length*.
- For example:
  - such as buffer completion flags or program status
  - send main storage addresses
  - function parameters
  - …

- Each MFC / SPE provides three mailbox queues:
  - Two outbound mailboxes to send messages from the SPE to the PPE or other SPEs
    - Only 1 entry available in each mailbox
      - SPU Write Outbound mailbox and SPU Write Outbound Interrupt mailbox
  - One Inbound mailboxes to send messages to the local SPE from the PPE or other SPEs.
    - 4 entries available
      - SPU Read Inbound mailbox
      - Each entry is a fullword.
Mailbox Example: Send EA of three arrays by mailbox (32-bit)

PPE Code

```
...  /* Create a spe context that we'll use to load our spe */
  threadData.context = spe_context_create(0, NULL);

  /* Load our spe program: array_spu_add*/
  spe_program_load( threadData.context,
                   &array_spu_add);

  /* Setup our SPE program parameters*/
  threadData.entry = SPE_DEFAULT_ENTRY;
  threadData.flags = 0;
  threadData.argp = NULL;
  threadData.envp = NULL;

  /* Now we can create thread to run on a SPE*/
  pthread_create( &threadData.pthread, NULL,
                  &ppu_pthread_function,
                  &threadData);

  // Setup mailbox messages (32-Bit ONLY)
  mbox_data[0] = (unsigned int)&array_a[0];
  mbox_data[1] = (unsigned int)&array_b[0];
  mbox_data[2] = (unsigned int)&array_c[0];

  while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[0];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
  while (retVal != 1);

  while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[1];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
  while (retVal != 1);

  while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[2];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
```

/* Send effective addresses of arrays to spe through mailboxes*/

```
while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[0];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
  }
while (retVal != 1);

while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[1];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
  }
while (retVal != 1);

while (spe_in_mbox_status(threadData.context) == 0);
  do
    mbox_dataPtr = &mbox_data[2];
    retVal = spe_in_mbox_write( threadData.context,
                                mbox_dataPtr, 1,
                                SPE_MBOX_ANY_NONBLOCKING);
```
Mailbox Example: Send EA of three arrays by mailbox (32-bit)

SPE Code

```c
#include <stdio.h>
#include <spu_mfcio.h>
#define ARRAY_SIZE 1024
#define MY_ALIGN(_my_var_def_, _my_al_) _my_var_def_ __attribute__((__aligned__(_my_al_)))
MY_ALIGN(float array_a[ARRAY_SIZE],128);
MY_ALIGN(float array_b[ARRAY_SIZE],128);
MY_ALIGN(float array_c[ARRAY_SIZE],128);

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp){
    int i;
    unsigned int Aaddr, Baddr, Caddr;

    /* First: Read mailbox 3 times to get our array effective addresses */
    Aaddr = (unsigned int) spu_read_in_mbox();
    Baddr = (unsigned int) spu_read_in_mbox();
    Caddr = (unsigned int) spu_read_in_mbox();

    /*Now that we have the array EAs we can DMA our data over */
    mfc_get(&array_a, Aaddr, (sizeof(float)*ARRAY_SIZE), 31, 0, 0);
    mfc_get(&array_b, Baddr, (sizeof(float)*ARRAY_SIZE), 31, 0, 0);
    mfc_write_tag_mask(1<<31);
    mfc_read_tag_status_all();

    /*array add*/
    for(i=0; i<ARRAY_SIZE; i++)
    {
        array_c[i] = array_a[i] + array_b[i];
    }

    /* Finally, DMA computed array back to main memory*/
    mfc_put(&array_c, Caddr, (sizeof(float)*ARRAY_SIZE), 31, 0, 0);
    mfc_write_tag_mask(1<<31);
    mfc_read_tag_status_all();
    return 0;
}
```

Mailbox Example:
Send EA of three arrays by mailbox (32-bit)
Signal notification

- SPE software can use polling or blocking when waiting a signal to appear, or it can set up interrupts to catch signals as they appear asynchronously.
- \textit{PPE sends signal-notification message to the SPE by writing a MMIO register in the SPE's MFC.}
- \textit{SPE can send a signal-notification message to another SPU using one of three special MFC commands: \texttt{sndsig, sndsigf and sndsigb}}

<table>
<thead>
<tr>
<th>SPE Channel #</th>
<th>Name</th>
<th>Channel Interface</th>
<th>MMIO Register Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mnemonic</td>
<td>Max. Entries</td>
</tr>
<tr>
<td>3</td>
<td>SPU Signal Notification 1</td>
<td>SPU_RdSigNotify1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>SPU Signal Notification 2</td>
<td>SPU_RdSigNotify2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SPU Configuration</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
MFC Synchronization Commands

MFC synchronization commands

- Used to control the order in which DMA storage accesses are performed
  - Four atomic commands (**getllar**, **putllc**, **putlluc**, and **putqlluc**),
  - Three send-signal commands (**sndsig**, **sndsigf**, and **sndsigb**), and
  - Three barrier commands (**barrier**, **mfcsync**, and **mfceieio**).

<table>
<thead>
<tr>
<th>Command</th>
<th>Supported By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>barrier</strong></td>
<td>PPE, SPE</td>
<td>Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the barrier command within the same command queue. The barrier command has no effect on the immediate DMA commands: <strong>getllar</strong>, <strong>putllc</strong>, and <strong>putqlluc</strong>.</td>
</tr>
<tr>
<td><strong>mfceieio</strong></td>
<td>PPE, SPE</td>
<td>Controls the ordering of get commands with respect to put commands, and of get commands with respect to get commands accessing storage that is caching inhibited and guarded. Also controls the ordering of put commands with respect to put commands accessing storage that is memory coherence required and not caching inhibited.</td>
</tr>
<tr>
<td><strong>mfcsync</strong></td>
<td>PPE, SPE</td>
<td>Controls the ordering of DMA put and get operations within the specified tag group with respect to other processing units and mechanisms in the system.</td>
</tr>
<tr>
<td><strong>sndsig</strong></td>
<td>PPE, SPE</td>
<td>Update SPU Signal Notification Registers in an I/O device or another SPE.</td>
</tr>
<tr>
<td><strong>sndsigb</strong></td>
<td>PPE, SPE</td>
<td>Update SPU Signal Notification Registers in an I/O device or another SPE with barrier.</td>
</tr>
<tr>
<td><strong>sndsigf</strong></td>
<td>PPE, SPE</td>
<td>Update SPU Signal Notification Registers in an I/O device or another SPE with fence.</td>
</tr>
</tbody>
</table>

1. There is a channel (for SPEs) and/or MMIO register (for PPE) to support the operation.
Special Notices -- Trademarks

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided “AS IS” with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Many of the features described in this document are operating system dependent and may not be available on Linux. For more information, please check: [http://www.ibm.com/systems/p/software/whitepapers/linux_overview.html](http://www.ibm.com/systems/p/software/whitepapers/linux_overview.html)

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

Revised January 19, 2006
Special Notices (Cont.) -- Trademarks


A full list of U.S. trademarks owned by IBM may be found at: http://www.ibm.com/legal/copytrade.shtml.

Cell Broadband Engine and Cell Broadband Engine Architecture are trademarks of Sony Computer Entertainment, Inc. in the United States, other countries, or both.
Rambus is a registered trademark of Rambus, Inc.
XDR and FlexIO are trademarks of Rambus, Inc.
UNIX is a registered trademark in the United States, other countries or both.
Linux is a trademark of Linus Torvalds in the United States, other countries or both.
Fedora is a trademark of Redhat, Inc.
Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.
Intel, Intel Xeon, Itanium and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.
AMD Opteron is a trademark of Advanced Micro Devices, Inc.
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.
TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).
SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECapc, SPECchpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corp (SPEC).
AltiVec is a trademark of Freescale Semiconductor, Inc.
PCI-X and PCI Express are registered trademarks of PCI SIG.
InfiniBand™ is a trademark the InfiniBand® Trade Association
Other company, product and service names may be trademarks or service marks of others.

Revised July 23, 2006
Special Notices - Copyrights

(c) Copyright International Business Machines Corporation 2005.  
All Rights Reserved.  Printed in the United States September 2005.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.  
IBM  IBM Logo  Power Architecture

Other company, product and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Microelectronics Division  
1580 Route 52, Bldg. 504  
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is http://www.chips.ibm.com