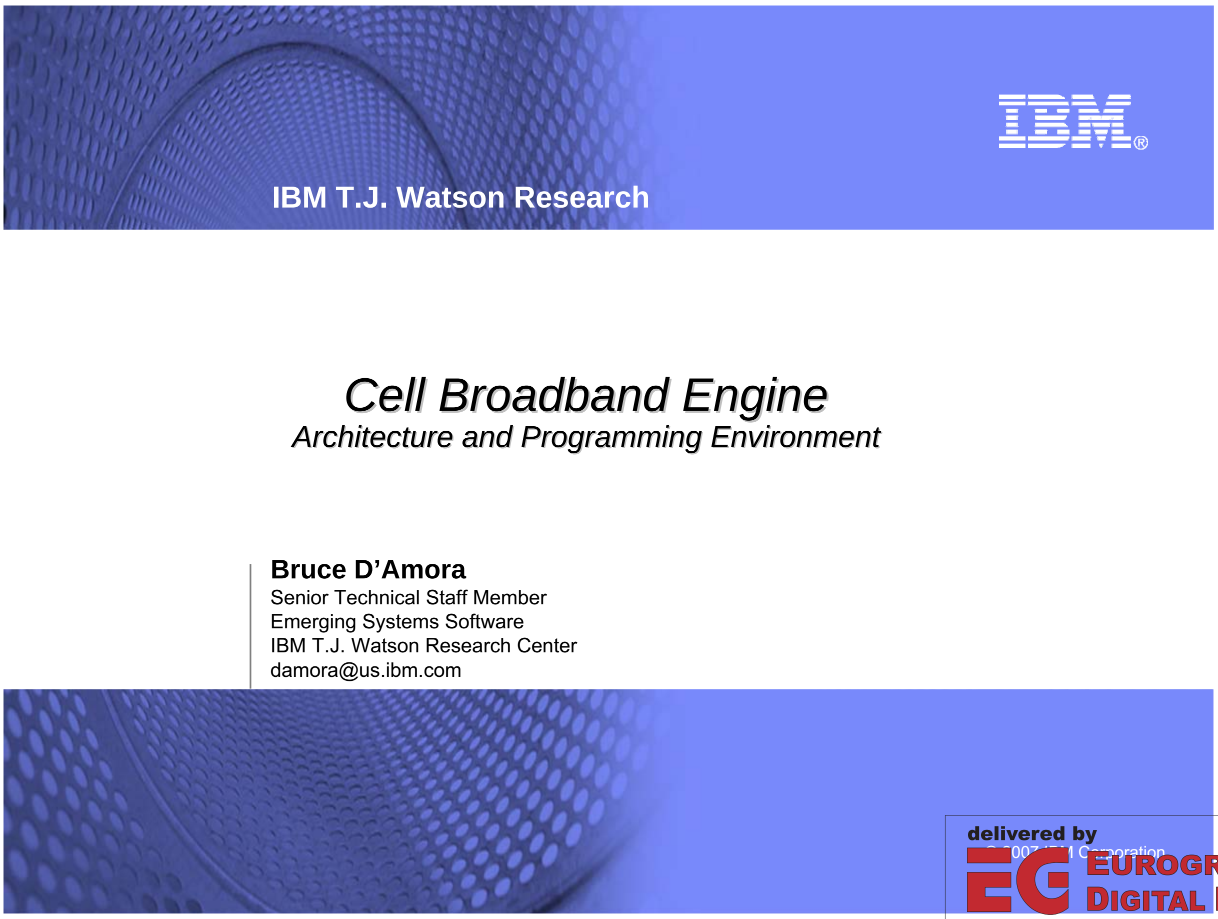
Programming the Cell BE for High Performance Graphics

Bruce D’Amora
IBM T.J. Watson Research Center

Michael McCool
RapidMind and University of Waterloo

Eurographics 2007
Tutorial Notes



IBM T.J. Watson Research

Cell Broadband Engine

Architecture and Programming Environment

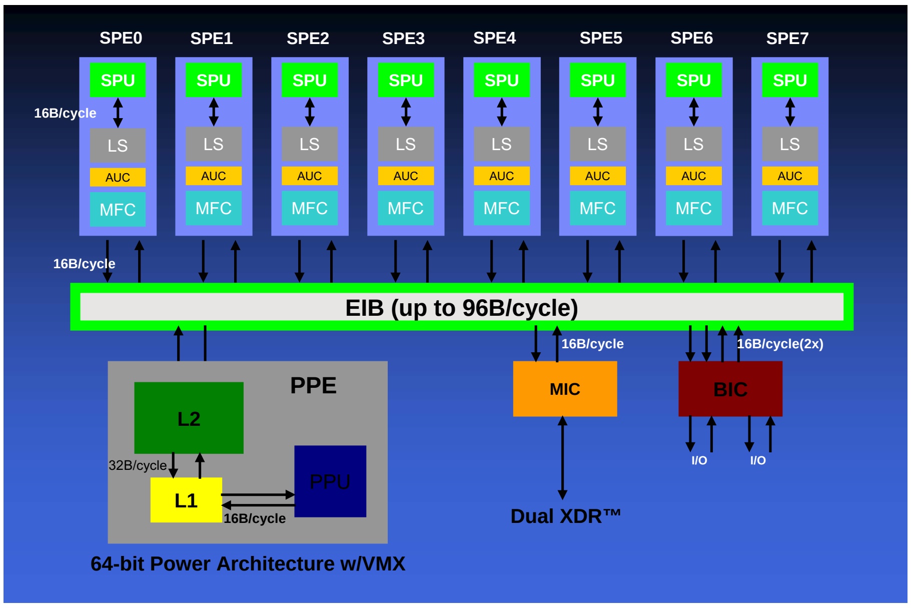
Bruce D’Amora
Senior Technical Staff Member
Emerging Systems Software
IBM T.J. Watson Research Center
damora@us.ibm.com

Agenda

- Architecture
- Programming Models
- Basic Programming
- Graphics Workloads
- Questions

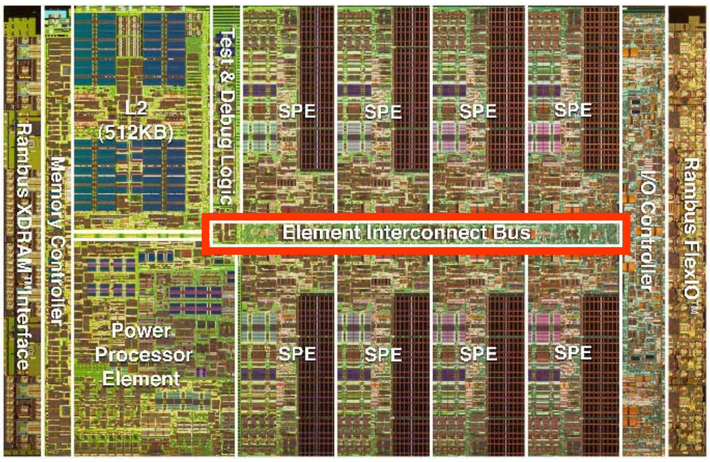
Architecture

Cell Broadband Engine Architecture



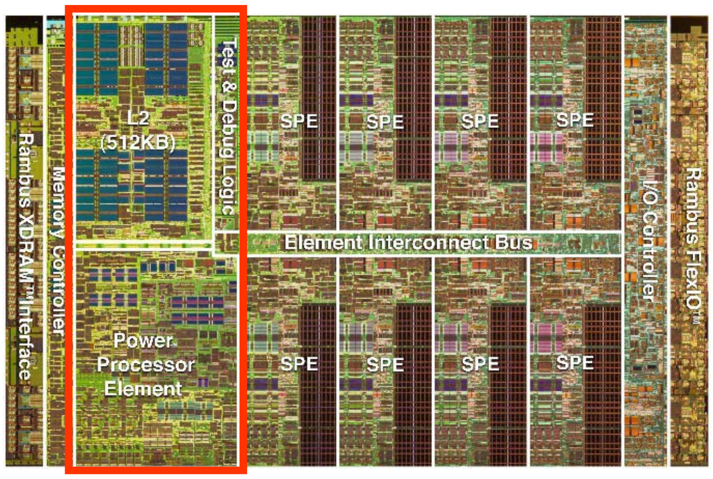
Element Interconnect Bus

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



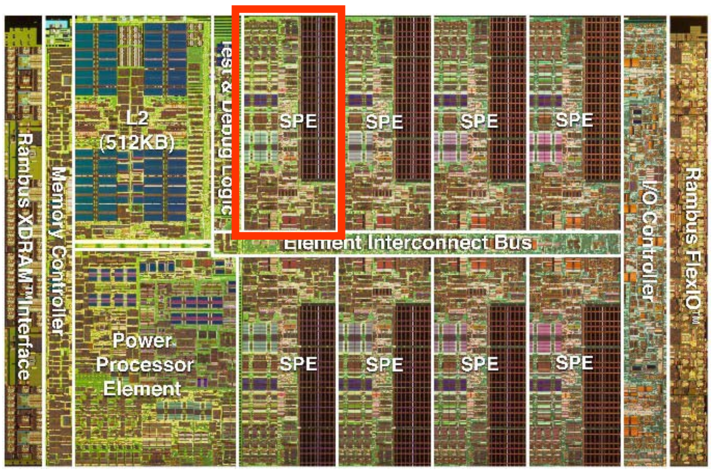
Power Processor Element

- PPE handles operating system and control tasks
- 64-bit Power Architecture™ with VMX
- In-order, 2-way hardware simultaneous multi-threading (SMT)
- Load/Store with 32KB L1 cache (I & D) and 512KB L2



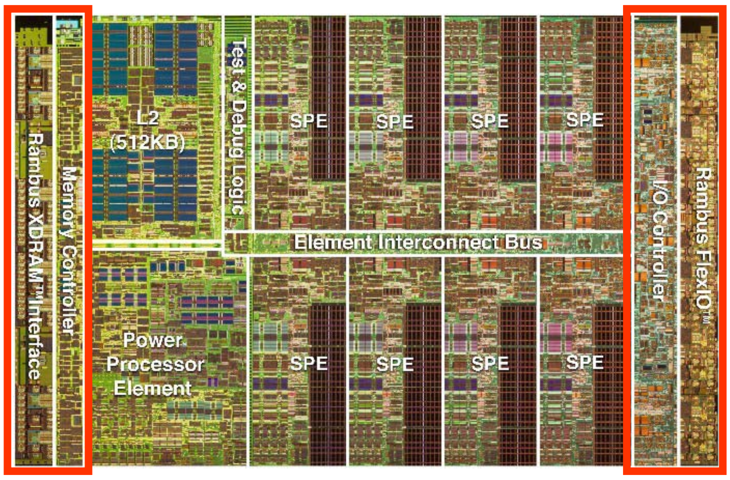
Synergistic Processor Element

- Dual issue, up to 16-way 128-bit SIMD
- Dedicated resources: 128 128-bit register file, 256KB Local Store
- Each can be dynamically configured to protect resources
- Dedicated DMA engine: Up to 16 outstanding requests per SPE



I/O and Memory Interfaces

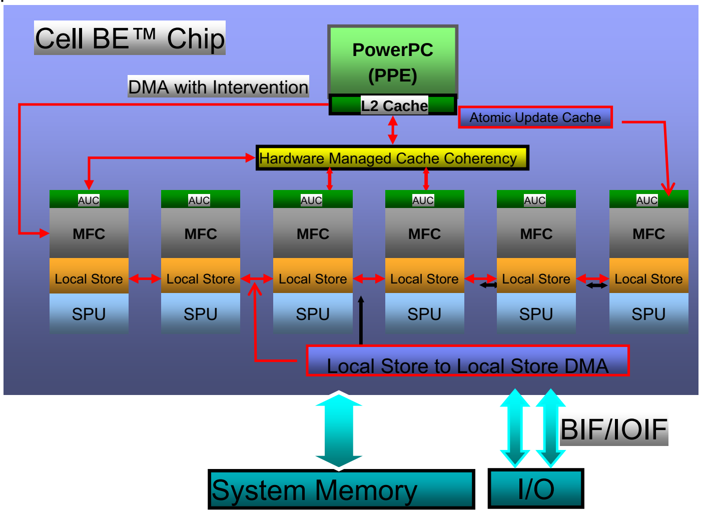
- Two configurable interfaces
- Up to 25.6 GB/s memory B/W
- Up to 70+ GB/s I/O B/W
  - Practical ~ 50GB/s



Programming

Cell BE Features Exploited by Software

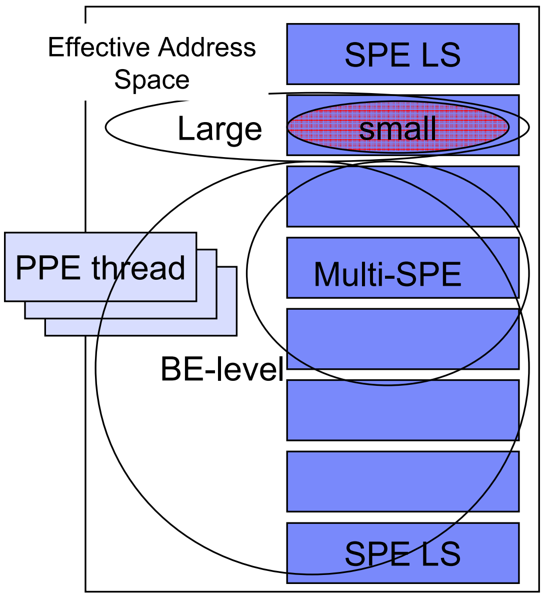
- Large register file
  - Keep intermediate and control data on chip
- DMA Engine – Memory Flow Controller
  - DMA between System Mem and LS
  - DMA from L2 cache-> LS
  - LS to LS DMA
  - Scatter->Gather support
- Atomic Update Cache
  - Implement synchronization commands
- SPE Signalling Registers
- SPE <-> PPE Mailboxes



- Resource Reservation and Allocation
  - PPE can be shared across logical partitions
  - SPEs can be assigned to logical partitions
  - SPEs independently or Group Allocated

Common Cell programming models

Single Cell environment:
- PPE programming models
- SPE Programming models
  - Small single-SPE models
  - Large single-SPE models
  - Multi-SPE parallel programming models



Small single-SPE models

- **Single tasked environment**
- **Small enough to fit into a 256KB- local store**
- **Sufficient for many dedicated workloads**
- **Two address spaces – (SPE) LS & (SPE/PPE) EA**
- **Explicit input and output of the SPE program**
  - DMA
  - Mailboxes
  - System calls

Small single-SPE models – tools and environment

- **SPE compiler/linker compiles and links an SPE executable**
- **The SPE executable image is embedded as reference-able RO data in the PPE executable**
- **A Cell programmer controls an SPE program via a PPE controlling process and its SPE management library**
  - i.e. loads, initializes, starts/stops an SPE program
- **The PPE controlling process, OS(PPE), and runtime(PPE or SPE) together establish the SPE runtime environment, e.g. argument passing, memory mapping, system call service.**

Small single-SPE models – PPE controlling program

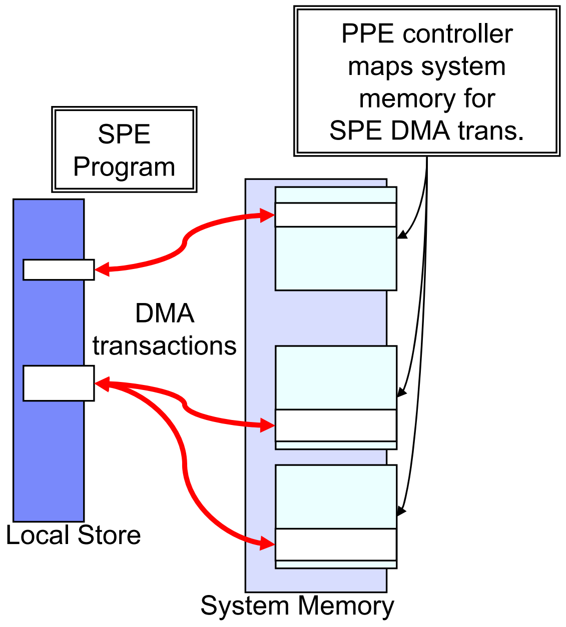
```
extern spe_program_handle spe_foo;  /* the spe image handle from CESOF */

int main()
{
            int rc, status;
            speid_t spe_id;

            /* load & start the spe_foo program on an allocated spe */
            spe_id = spe_create_thread (0, &spe_foo, 0, NULL, -1, 0);

            /* wait for spe prog. to complete and return final status */
            rc = spe_wait (spe_id, &status, 0);

            return status;
}
```

Small single-SPE models – SPE code

```
/* spe_foo.c: A C program to be compiled into an executable called "spe_foo" */

  int main( int speid, addr64 argp, addr64 envp)
  {
            char i;

            /* do something intelligent here */
            i = func_foo (argp);

            /* when the syscall is supported */
            printf( "Hello world! my result is %d \n", i);

            return i;
  }
```

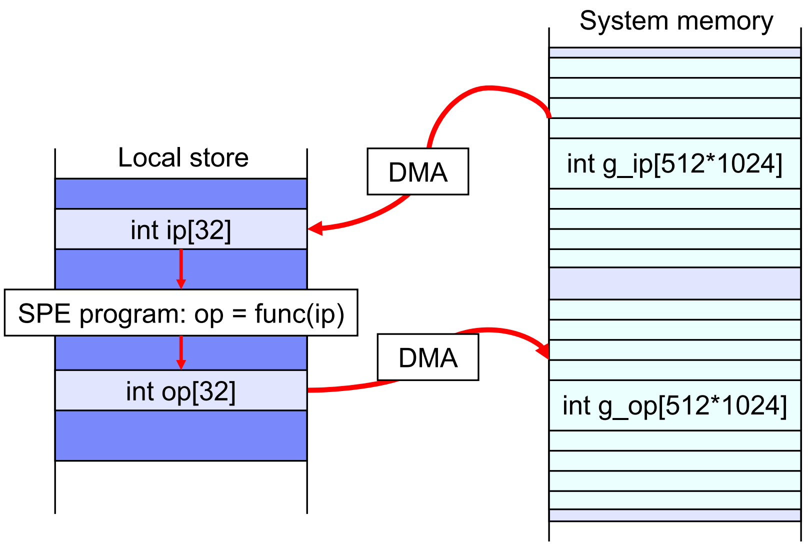
Large single-SPE programming models

- Data or code working set cannot fit completely into a local store
- The PPE controlling process, kernel, and libspe runtime set up the system memory mapping as SPE's secondary memory store
- The SPE program accesses the secondary memory store via its software-controlled SPE DMA engine - Memory Flow Controller (MFC)



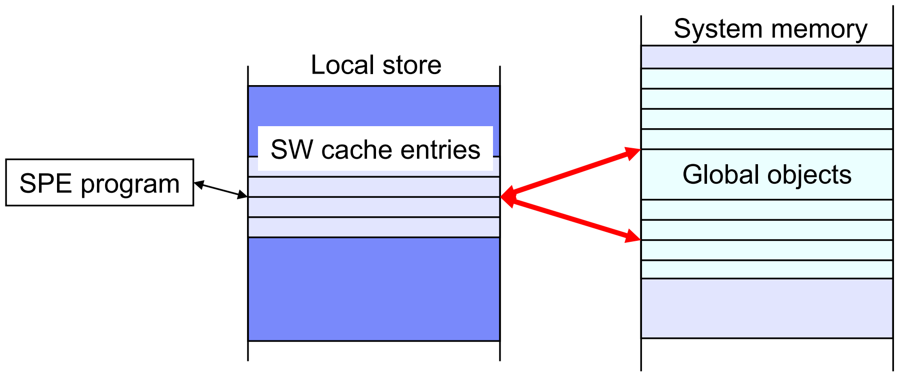
Large single-SPE programming models – I/O data

- System memory for large size input / output data
  - e.g. Streaming model



Large single-SPE programming models-SW Cache

- System memory as secondary memory store
  - Manual management of data buffers
  - Automatic software-managed data cache
    - Software cache framework libraries
    - Compiler runtime support



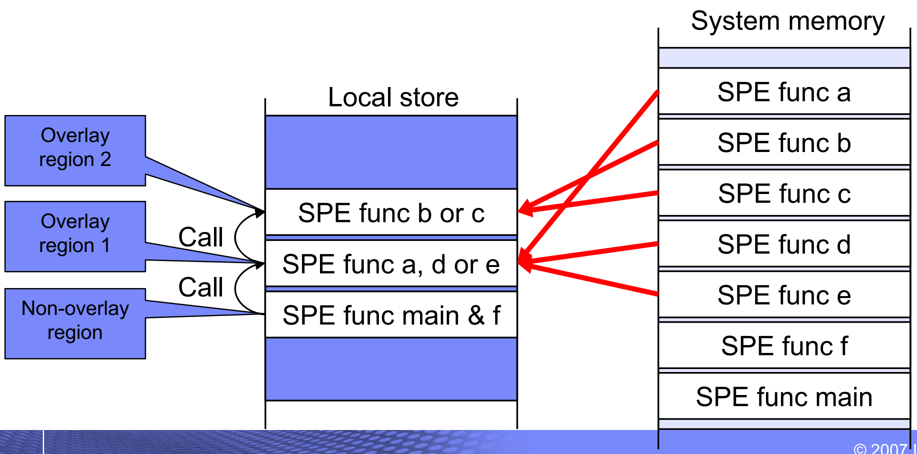
Shared-memory Multiprocessor

- Cell BE can be programmed as a shared-memory multiprocessor
  - PPE and SPE have different instruction sets and compilers
- SPEs and the PPE fully inter-operate in a cache-coherent model
- Cache-coherent DMA operations for SPEs
  - DMA operations use effective address common to all PPE and SPEs
  - SPE shared-memory ***store*** instructions are replaced
    - A store from the register file to the LS
    - DMA operation from LS to shared memory
  - SPE shared-memory ***load*** instructions are replaced
    - DMA operation from shared memory to LS
    - A load from LS to register file
- A compiler could manage part of the LS as a local cache for instructions and data obtained from shared memory.

Large single-SPE programming models- Overlays

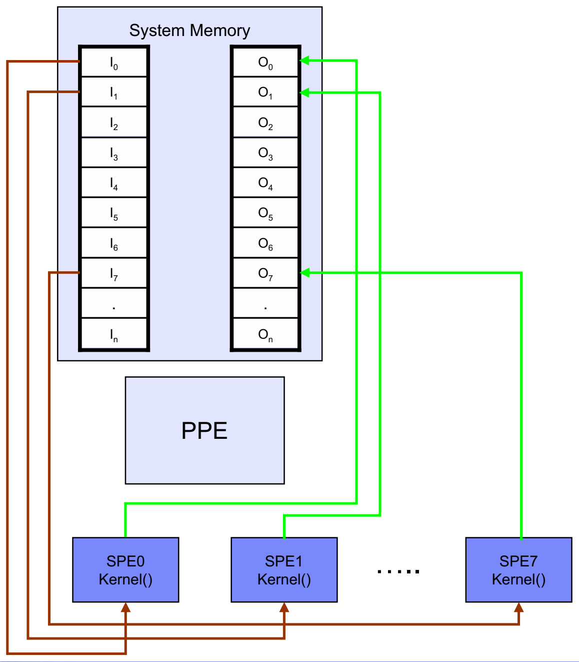
- System memory as secondary memory store
  - Manual loading of plug-in into code buffer
    - Plug-in framework libraries
  - Automatic and manual software-managed code overlay
    - Compiler and Linker generated overlaying code

An overlay is SPU code that is dynamically loaded and executed by a running SPU program. It cannot be independently loaded or run on an SPE



Parallel programming models – Streaming

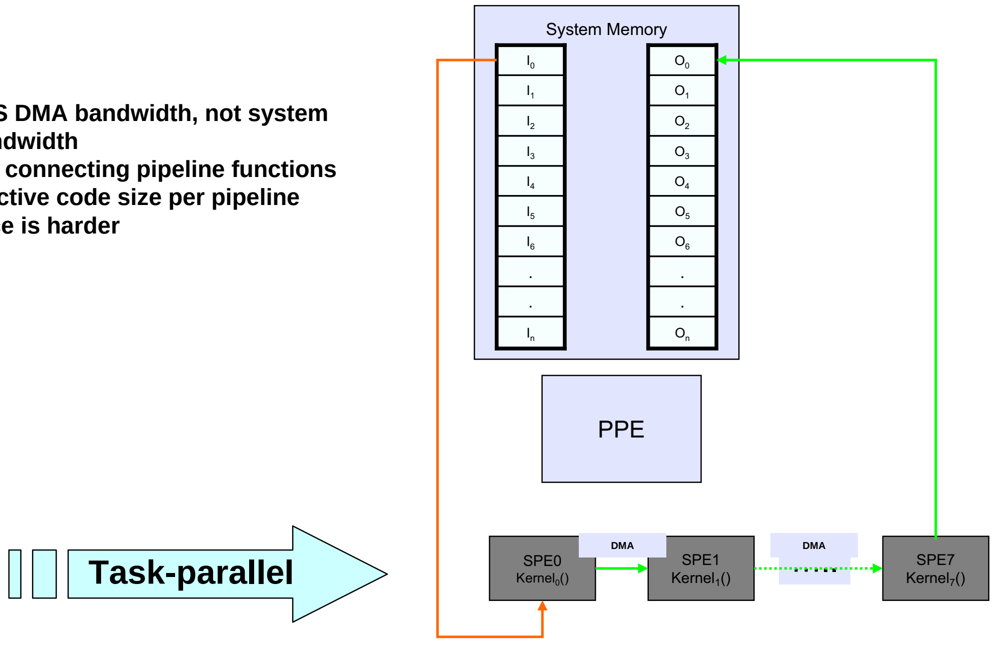
- **SPE initiated DMA**
- **Large array of data fed through a group of SPE programs**
- **A special case of job queue with regular data**
- **Each SPE program locks on the shared job queue to obtain next job**
- **For uneven jobs, workloads are self-balanced among available SPEs**



Data-parallel

Parallel programming models – Pipeline

- Use LS to LS DMA bandwidth, not system memory bandwidth
- Flexibility in connecting pipeline functions
- Larger collective code size per pipeline
- Load-balance is harder


System Memory
I_0 I_1 I_2 I_3 I_4 I_5 I_6 . . I_n
O_0 O_1 O_2 O_3 O_4 O_5 O_6 . . O_n
PPE
SPE0 $Kernel_0()$
DMA
SPE1 $Kernel_1()$
DMA
SPE7 $Kernel_7()$
Task-parallel

Programming Model Final Points

- A proper programming model reduces development cost while achieving higher performance
- Programming frameworks and abstractions help with productivity
- Mixing programming models are common practice
- New models may be developed for particular applications.
- With the vast computational capacity, it is not hard to achieve a performance gain from an existing legacy base
  - Top performance is harder
- Tools are critical in improving programmer productivity

Basic Programming

“Hello World!” – SPE Only

- SPU Program

```
#include <stdio.h>

int main()
{
        printf("Hello world!\n");
        return 0;
}
```

- SPU Makefile

PROGRAM_spu tells make to use SPE compiler

```
PROGRAM_spu   := hello_spu
include $(CELL_TOP)/make.footer
```

Synergistic PPE and SPE (SPE Embedded)

- **Applications use software constructs called SPE contexts to manage and control SPEs.**
- **Linux schedules SPE contexts from all running applications onto the physical SPE resources in the system for execution according to the scheduling priorities and policies associated with the runable SPE contexts.**
- **libspe provides API for communication and data transfer between PPE threads and SPEs.**

How a PPE program embeds an SPE program?

4 basic steps must be done by the PPE program

1. Create an SPE context.
2. Load an SPE executable object into the SPE context local store.
3. Run the SPE context. This transfers control to the operating system, which requests the actual scheduling of the context onto a physical SPE in the system.
4. Destroy the SPE context.

SPE context creation

- spe_context_create **- Create and initialize a new SPE context data structure.**

```
#include <libspe2.h>
spe_context_ptr_t spe_context_create(unsigned int flags,
spe_gang_context_ptr_t gang)
```

  – *`flags`* - A bit-wise OR of modifiers that are applied when the SPE context is created.
  – *`gang`* - Associate the new SPE context with this gang context. If NULL is specified, the new SPE context is not associated with any gang.
  – On success, a pointer to the newly created SPE context is returned.

spe_program_load

- spe_program_load **- Load an SPE main program.**

```
#include <libspe2.h>
int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t
*program)
```

  – *`spe`* - A valid pointer to the SPE context for which an SPE program should be loaded.
  – *`program`* - A valid address of a mapped SPE program.

spe_context_run

- spe_context_run **- Request execution of an SPE context.**

```
#include <libspe2.h>
int spe_context_run(spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void
*argp, void *envp, spe_stop_info_t *stopinfo)
```

- `spe` - A pointer to the SPE context that should be run.
- `entry` - Input: The entry point, that is, the initial value of the SPU instruction pointer, at which the SPE program should start executing. If the value of entry is SPE_DEFAULT_ENTRY, the entry point for the SPU main program is obtained from the loaded SPE image. This is usually the local store address of the initialization function crt0.
- `runflags` - A bit mask that can be used to request certain specific behavior for the execution of the SPE context. 0 indicates default behavior.
- `argp` - An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program,
- `envp` - An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program,
- `stopinfo` An (optional) pointer to a structure of type spe_stop_info_t

spe_context_destroy

- spe_context_destroy **- Destroy the specified SPE context.**

```
#include <libspe2.h>
int spe_context_destroy (spe_context_ptr_t spe)
```

- `spe` - Specifies the SPE context to be destroyed
- On success, **0** (zero) is returned, else -1 is returned

“Hello World!” – SPE object embedded in PPE program

- SPU program
  - Same as for SPE only
- SPU Makefile

```
PROGRAM_spu   := hello_spu
LIBRARY_embed  := hello_spu.a
include $(CELL_TOP)/make.footer
```

“Hello World!” – PPU program

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

extern spe_program_handle_t hello_spu;

int main(void)
{
   spe_context_ptr_t speid;
   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("spe_context_create");
   return -2;
   }
```

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

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

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

PPU Makefile

```
DIRS = spu
PROGRAM_ppu = hello_be1
IMPORTS = spu/hello_spu.a –lspe2 -lpthread
include $(CELL_TOP)/make.footer
```

PPE and SPE Synergistic Programming

PPU Code

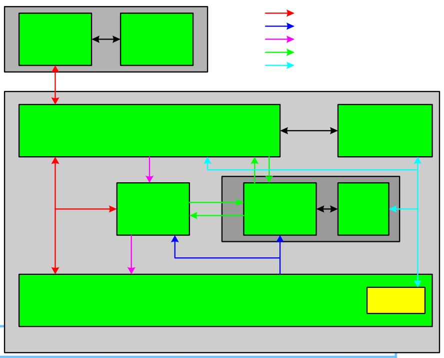
```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
extern spe_program_handle_t hello_spu;
int main(void)
{
          . . . . .
          // Run the SPE context
          rc = spe_context_run(speid, &entry, 0, argp, envp, &stop_info);
          . ....
}
```

SPU Code

```
#include <stdio.h>

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

Primary Communication Mechanisms

- **DMA transfers**
- **Mailbox messages**
- **Signal-notification**
- **All three are implemented and controlled by the SPE's MFC**



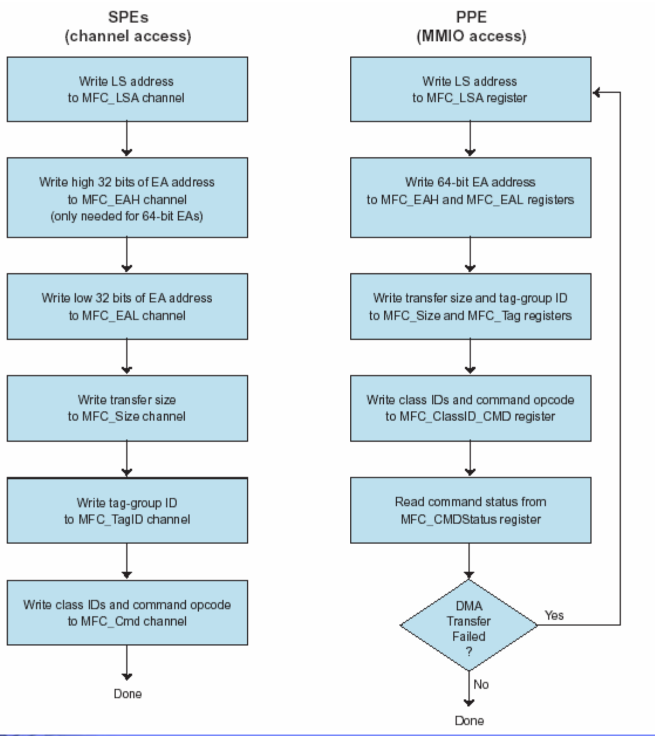
Mechanism	
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called *signaling)* uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

Memory Flow Controller (MFC) Commands

- **Main mechanism for SPUs to**
  - access main storage (DMA commands)
  - maintain **synchronization** with other processors and devices in the system (Synchronization commands)
- **Can be issued either by SPU via its MFC or by PPE or other device, as follows:**
  - Code running on the SPU issues an MFC command by executing a series of writes and/or reads using **channel instructions -** read channel (rdch), write channel (wrch), and read channel count (rchcnt).
  - Code running on the PPE or other devices issues an MFC command by performing a series of stores and/or loads to **memory-mapped I/O** (MMIO) registers in the MFC
- **MFC commands are queued in one of two independent MFC command queues:**
  - MFC SPU Command Queue — For channel-initiated commands by the associated SPU
  - MFC Proxy Command Queue — For MMIO-initiated commands by the PPE or other device

Sequences for Issuing MFC Commands

- All operations on a given channel are unidirectional
  - only read or write operations for a given channel, not bidirectional
- Accesses to channel-interface resources through MMIO addresses do not stall
- Channel operations are done in program order
- Channel read operations to reserved channels return ‘0’s
- Channel write operations to reserved channels have no effect
- Reading of channel counts on reserved channels returns ‘0’
- Channel instructions use the 32-bit preferred slot in a 128-bit transfer

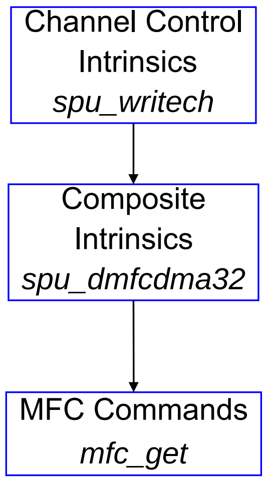

SPEs (channel access)
Write LS address to MFC_LSA channel
Write high 32 bits of EA address to MFC_EAH channel (only needed for 64-bit EAs)
Write low 32 bits of EA address to MFC_EAL channel
Write transfer size to MFC_Size channel
Write tag-group ID to MFC_TagID channel
Write class IDs and command opcode to MFC_Cmd channel
Done
PPE (MMIO access)
Write LS address to MFC_LSA register
Write 64-bit EA address to MFC_EAH and MFC_EAL registers
Write transfer size and tag-group ID to MFC_Size and MFC_Tag registers
Write class IDs and command opcode to MFC_ClassID_CMD register
Read command status from MFC_CMDStatus register
DMA Transfer Failed ?
Yes
No
Done

DMA Overview

DMA Commands

- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**

DMA Commands


Channel Control Intrinsics *spu_writech*
Composite Intrinsics *spu_dmfcdma32*
MFC Commands *mfc_get*

defined as macros in
spu_mfcio.h

For details see: SPU C/C++ Language Extensions

DMA Get and Put Command (SPU)

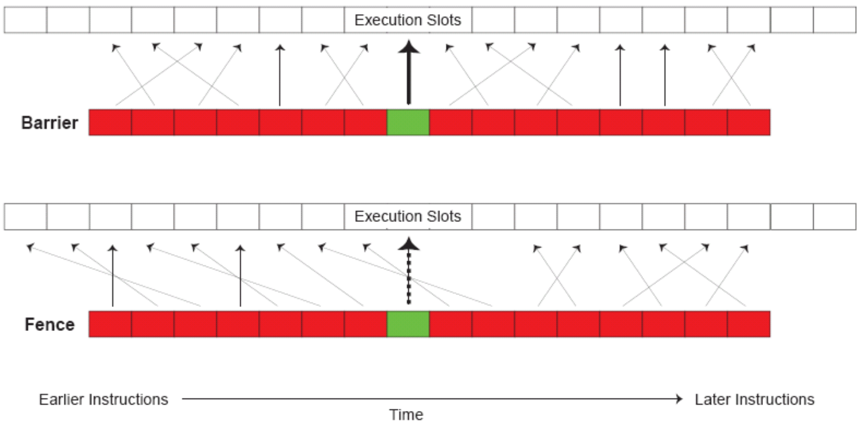
- DMA get from main memory into local store
  (void) mfc_get(volatile void *ls, uint64_t ea, uint32_t size,
  uint32_t tag, uint32_t tid, uint32_t rid)
- DMA put into main memory from local store
  (void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size,
  uint32_t tag, uint32_t tid, uint32_t rid)
- **To ensure order of DMA request execution:**
  – mfc_putf : **fenced** (all commands executed before within the same tag group must finish first, later ones could be before)

  – mfc_putb : **barrier** (the barrier command and all commands issued thereafter are not executed until all previously issued commands in the same tag group have been performed)

DMA-Command Tag Groups

- **5-bit DMA Tag for all DMA commands (except getllar, putllc, and putlluc)**
- **Tag can be used to**
  - determine status for entire group or command
  - check or wait on the completion of all queued commands in one or more tag groups
- **Tagging is optional but can be useful when using barriers to control the ordering of MFC commands within a single command queue.**
- **Synchronization of DMA commands within a tag group: fence and barrier**
  - Execution of a fenced command option is delayed until all previously issued commands within the same tag group have been performed.
  - Execution of a barrier command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.

Barriers and Fences



🟩 Synchronizing command
🟥 Non-synchronizing command
□ Execution slot

DMA Characteristics

- DMA transfers
  - transfer sizes can be 1, 2, 4, 8, and n*16 bytes (n integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable (cache-line)
- DMA command queues per SPU
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
  - ***SPU-initiated DMA is always preferable***
- DMA tags
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands
- DMA lists
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests

PPE – SPE DMA Transfer

Transfer from PPE (Main Memory) to SPE

- DMA get from main memory

  mfc_get(lsaddr, ea, size, tag_id, tid, rid);
  - lsaddr = target address in SPU local store for fetched data (SPU local address)
  - ea = effective address from which data is fetched (global address)
  - size = transfer size in bytes
  - tag_id = tag-group identifier
  - tid = transfer-class id
  - rid = replacement-class id
- Also available via “composite intrinsic”:

  spu_mfcdma64(lsaddr, eahi, ealow, size, tag_id, cmd);

DMA Command Status (SPE)

- DMA read and write commands are non-blocking
- Tags, tag groups, and tag masks used for:
  - checking status of DMA commands
  - waiting for completion of DMA commands
- Each DMA command has a 5-bit tag
  - commands with same tag value form a “tag group”
- Tag mask is used to identify tag groups for status checks
  - tag mask is a 32-bit word
  - each bit in the tag mask corresponds to a specific tag id:

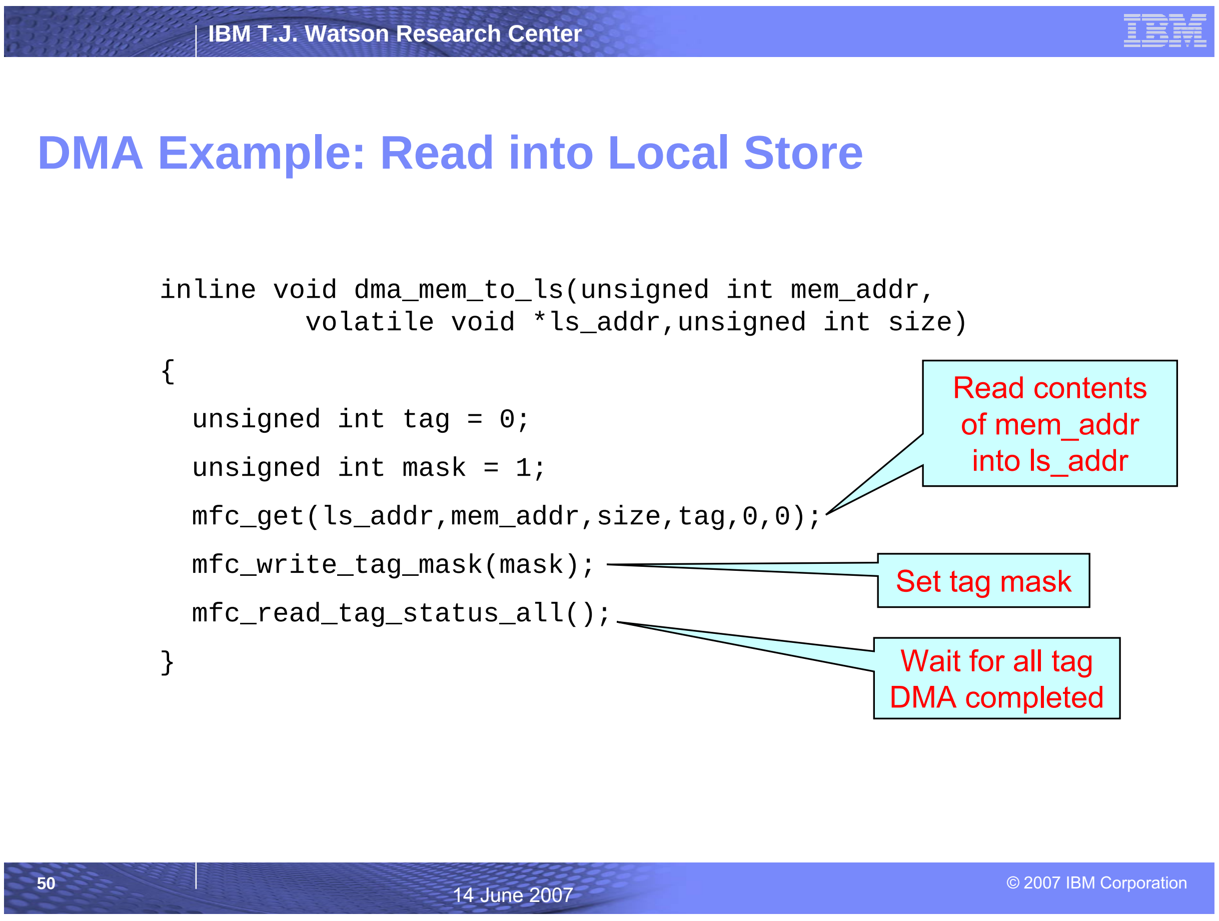
    tag_mask = (1 << tag_id)

DMA Tag Status (SPE)

- Set tag mask
  unsigned int tag_mask;
  mfc_write_tag_mask(tag_mask);
  – tag mask remains set until changed
- Fetch tag status
  unsigned int result;
  result = mfc_read_tag_status();   /* or mfc_stat_tag_status(); */
  – tag status is logically ANDed with current tag mask
  – tag status bit of ‘1’ indicates that no DMA requests tagged with the specific tag id (corresponding to the status bit location) are still either in progress or in the DMA queue

Waiting for DMA Completion (SPE)

- Wait for any tagged DMA:
  – mfc_read_tag_status_any():
  – wait until **any** of the specified tagged DMA commands is completed
- Wait for all tagged DMA:
  – mfc_read_tag_status_all():
  – wait until **all** of the specified tagged DMA commands are completed
- Specified tagged DMA commands = command specified by current tag mask setting



IBM T.J. Watson Research Center

DMA Example: Read into Local Store

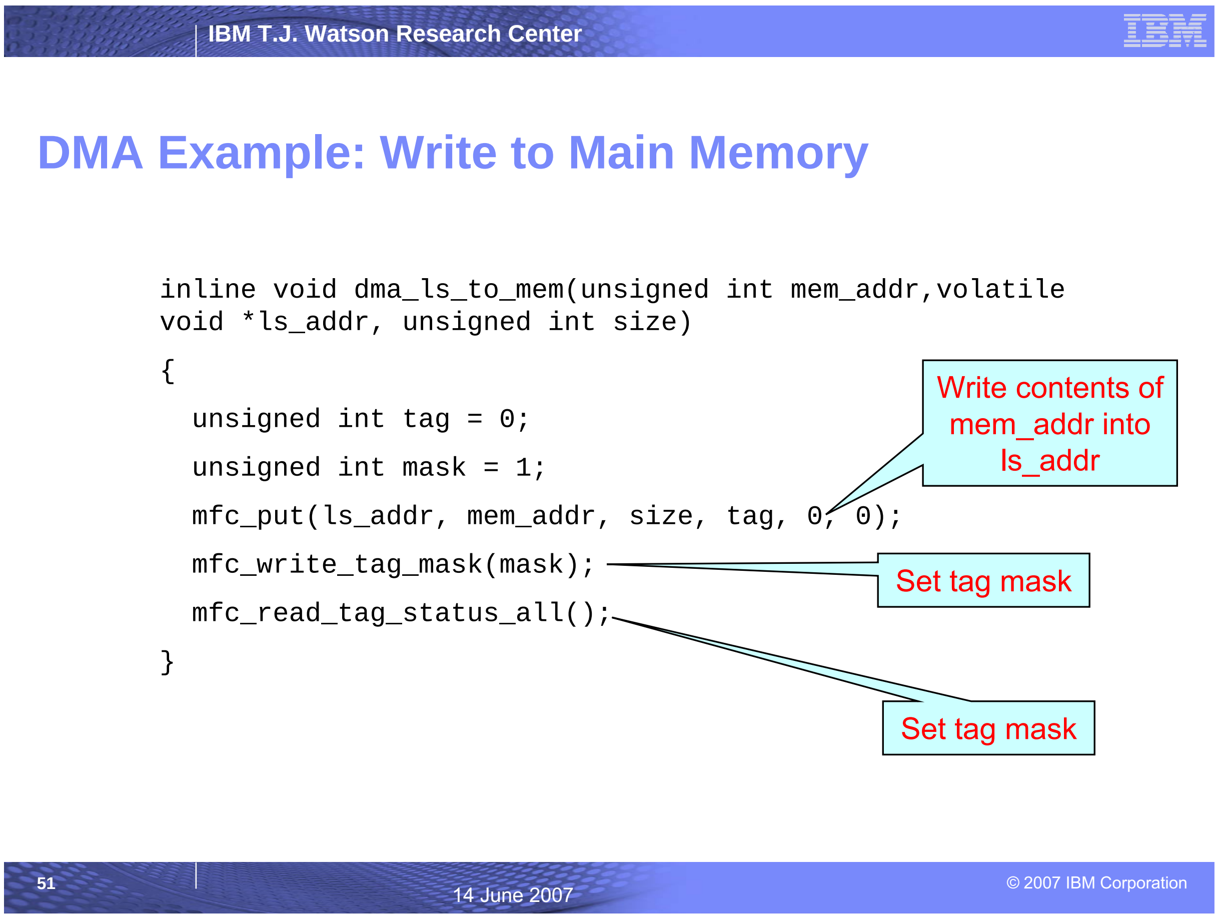
```
inline void dma_mem_to_ls(unsigned int mem_addr,
         volatile void *ls_addr,unsigned int size)
{
  unsigned int tag = 0;
  unsigned int mask = 1;
  mfc_get(ls_addr,mem_addr,size,tag,0,0);
  mfc_write_tag_mask(mask);
  mfc_read_tag_status_all();
}
```

Read contents of mem_addr into ls_addr

Set tag mask

Wait for all tag DMA completed

50 14 June 2007 © 2007 IBM Corporation



IBM T.J. Watson Research Center

DMA Example: Write to Main Memory

```
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();
}
```

Write contents of mem_addr into ls_addr

Set tag mask

Set tag mask

51 14 June 2007 © 2007 IBM Corporation

SPE – SPE DMA Transfer

SPE – SPE DMA

- **Address in the other SPE's local store is represented as a 32-bit effective address (global address)**
- **SPE issuing the DMA command needs a pointer to the other SPE's local store as a 32-bit effective address (global address)**
- **PPE code can obtain effective address of an SPE's local store:**

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

- **Effective address of an SPE's local store can then be made available to other SPEs (e.g. via DMA or mailbox)**

DMA support for Double Buffering

```
#include <spu_intrinsics.h>
#include "cbe_mfc.h"
#define BUFFER_SIZE 4096
volatile unsigned char B[2][BUFFER_SIZE] __attribute__ ((aligned(128)));
void double_buffer_example (unsigned int eahi, unsigned int ealow, int buffers)
{
        int next_idx, buf_idx = 0;
        // Initiate first DMA transfer using first buffer
        spu_mfcdma64(B[buf_idx], eahi, ealow, BUFFER_SIZE, buf_idx, MFC_GET_CMD);
        ealow += BUFFER_SIZE;
        while (--buffers) {
                next_idx = buf_idx ^ 1;
                // Initiate next DMA transfer
                spu_mfcdma64(B[next_idx], eahi, ealow, BUFFER_SIZE, next_idx, MFC_GET_CMD);
                ealow += BUFFER_SIZE;
                // Wait for previous transfer to complete
                spu_writech (MFC_WrTagMask, 1 << buf_idx);
                (void) spu_mfcstat(2);
                // Use the data from the previous transfer
                use_data (B[buf_idx]);
                buf_idx = next_idx;
        }
        // Wait for last transfer to complete
        spu_writech (MFC_WrTagMask, 1 << buf_idx);
        (void)spu_mfcstat(2);
        // Use the data from the last transfer
        use_data (B[buf_idx]);
}
```

Tips to Achieve Peak Bandwidth for DMAs

- The performance of a DMA data transfer is best when the source and destination addresses have the same quadword offsets within a PPE cache line.
- Quadword-offset-aligned data transfers generate full cache-line bus requests for every unrolling, except possibly the first and last unrolling.
- Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively.

Mailboxes Overview

Uses of Mailboxes

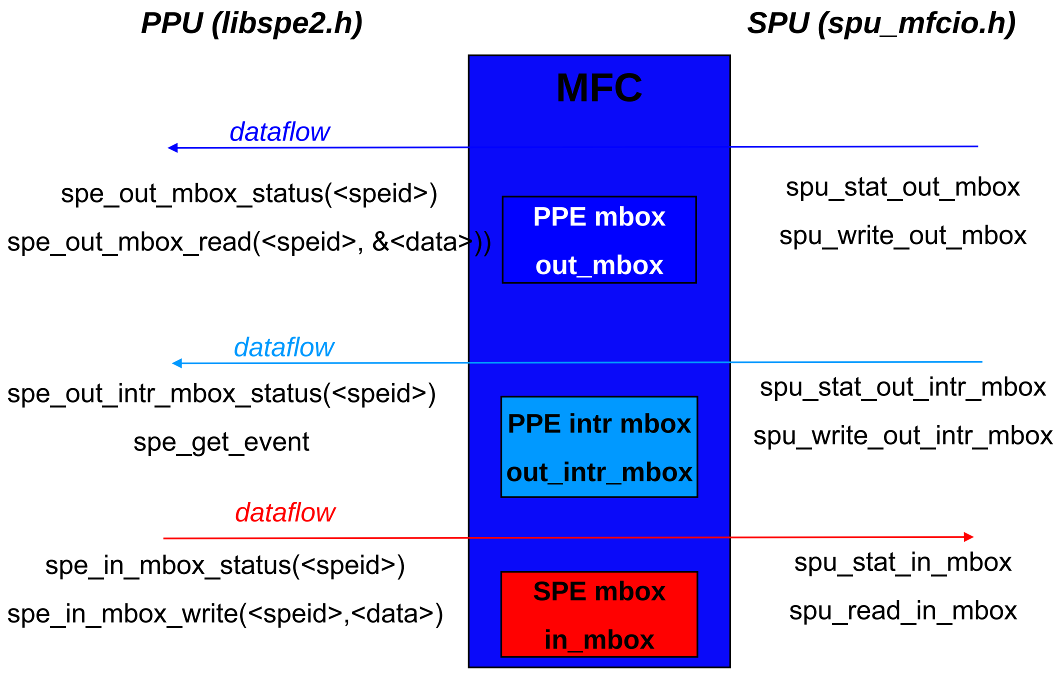
- **To communicate messages up to 32 bits in length, such as buffer completion flags or program status**
  - e.g., When the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then writes to an outbound mailbox to notify the PPE that its computation is complete
- **Can be used for any short-data transfer purpose, such as sending of storage addresses, function parameters, command parameters, and state-machine parameters**
- **Can also be used for communication between an SPE and other SPEs, processors, or devices**
  - Privileged software needs to allow one SPE to access the mailbox register in another SPE by mapping the target SPE's problem-state area into the EA space of the source SPE.
  - If software does not allow this, then only atomic operations and signal notifications are available for SPE-to-SPE communication.

Mailboxes - Characteristics

Each MFC provides three mailbox queues of 32 bit each:

- PPE (“SPU write outbound”) mailbox queue
  - SPE writes, PPE reads
  - 1 entry per queue
  - SPE stalls writing to full mailbox
- PPE (“SPU write outbound”) interrupt mailbox queue
  - like PPE mailbox queue, but an interrupt is posted to the PPE when the mailbox is written
- SPU (“SPU read inbound”) mailbox queue
  - PPE writes, SPE reads
  - 4 entries per queue
  - can be overwritten

Mailboxes API – libspe2



SPU Write Outbound Mailboxes

SPU Write Outbound Mailbox

- The value **written** to the SPU Write Outbound Mailbox channel SPU_WrOutMbox is entered into the outbound mailbox in the MFC if the mailbox has capacity to accept the value.
- If the mailbox can **accept** the value, the channel count for SPU_WrOutMbox is **decremented** by ‘1’.
- If the outbound mailbox is **full**, the channel count will read as ‘0’.
- If SPE software writes a value to SPU_WrOutMbox when the channel count is ‘0’, the SPU will **stall** on the write.
- The SPU **remains stalled** until the PPE or other device reads a message from the outbound mailbox by reading the MMIO address of the mailbox.
- When the mailbox is **read** through the MMIO address, the channel count is incremented by ‘1’

SPU Write Outbound Interrupt Mailbox

- The value **written** to the SPU Write Outbound Interrupt Mailbox channel (SPU_WrOutIntrMbox) is entered into the outbound interrupt mailbox if the mailbox has capacity to accept the value.
- If the mailbox can **accept** the message, the channel count for SPU_WrOutIntrMbox is decremented by ‘1’, and an **interrupt is raised** in the PPE or other device, depending on interrupt enabling and routing.
- There is no ordering of the interrupt and previously issued MFC commands.
- If the outbound interrupt mailbox is **full**, the channel count will read as ‘0’.
- If SPE software writes a value to SPU_WrOutIntrMbox when the channel count is ‘0’, the SPU will **stall** on the write.
- The SPU **remains stalled** until the PPE or other device reads a mailbox message from the outbound interrupt mailbox by reading the MMIO address of the mailbox.
- When this is done, the channel count is **incremented** by ‘1’.

Waiting to Write SPU Write Outbound Mailbox Data

- **To avoid SPU stall, SPU can use the read-channel-count instruction on the SPU Write Outbound Mailbox channel to determine if the queue is empty before writing to the channel.**
- **If the read-channel-count instruction returns ‘0’, the SPU Write Outbound Mailbox Queue is full.**
- **If the read channel-count instruction returns a non-zero value, the value indicates the number of free entries in the SPU Write Outbound Mailbox Queue.**
- **When the queue has free entries, the SPU can write to this channel without stalling the SPU.**

Polling SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox.

```
/* To write the value 1 to the SPU Write Outbound Interrupt Mailbox instead
* of the SPU Write Outbound Mailbox, simply replace SPU_WrOutMbox
* with SPU_WrOutIntrMbox in the following example.*/
unsigned int mb_value;
do {
/* Do other useful work while waiting.*/
} while (!spu_readchcnt(SPU_WrOutMbox));   // 0 → full, so something useful
spu_writech(SPU_WrOutMbox, mb_value);
```

Polling for or Block on an SPU Write Outbound Mailbox Available Event

```
#define MBOX_AVAILABLE_EVENT 0x00000080
unsigned int event_status;
unsigned int mb_value;
spu_writech(SPU_WrEventMask, MBOX_AVAILABLE_EVENT);
do {
    /*
    * Do other useful work while waiting.
    */
} while (!spu_readchcnt(SPU_RdEventStat));
event_status = spu_readch(SPU_RdEventStat); /* read status */
spu_writech(SPU_WrEventAck, MBOX_AVAILABLE_EVENT); /* acknowledge event */
spu_writech(SPU_WrOutMbox, mb_value); /* send mailbox message */
```

- NOTES: To block, instead of poll, simply delete the do-loop above.

PPU reads SPU Outbound Mailboxes

- PPU must check Mailbox Status Register first
  - check that unread data is available in the SPU Outbound Mailbox or SPU Outbound Interrupt Mailbox
  - otherwise, stale or undefined data may be returned
- To determine that unread data is available
  - PPE reads the Mailbox Status register
  - extracts the count value from the SPU_Out_Mbox_Count field
- count is
  - non-zero → at least one unread value is present
  - zero → PPE should not read but poll the Mailbox Status register

SPU Read Inbound Mailbox

SPU Read Inbound Mailbox Channel

- **Mailbox is FIFO queue**
  - If the SPU Read Inbound Mailbox channel (SPU_RdInMbox) has a message, the value read from the mailbox is the oldest message written to the mailbox.
- **Mailbox Status (empty: channel count =0)**
  - If the inbound mailbox is empty, the SPU_RdInMbox channel count will read as '0'.
- **SPU stalls on reading empty mailbox**
  - If SPE software reads from SPU_RdInMbox when the channel count is '0', the SPU will stall on the read. The SPU remains stalled until the PPE or other device writes a message to the mailbox by writing to the MMIO address of the mailbox.
- **When the mailbox is written through the MMIO address, the channel count is incremented by '1'.**
- **When the mailbox is read by the SPU, the channel count is decremented by '1'.**

SPU Read Inbound Mailbox Characteristics

- **The SPU Read Inbound Mailbox can be overrun by a PPE in which case, mailbox message data will be lost.**
- **A PPE writing to the SPU Read Inbound Mailbox will not stall when this mailbox is full.**

PPE Access to Mailboxes

- PPE can derive “addresses” of mailboxes from spe thread id
- First, create SPU thread, e.g.:

```
speid_t spe_id;
spe_id = spe_create_thread(0,spu_load_image,NULL,NULL,-1,0);
```

  – spe_id has type speid_t (normally an int)
- PPE mailbox calls use spe_id to identify desired SPE’s mailbox
- Functions are in libspe.a

Read: PPE Mailbox Queue – PPE Calls (libspe.h)

- “SPU outbound” mailbox
- Check mailbox status:

```
unsigned int count;
count = spe_stat_out_mbox(spe_id);
```

  – count = 0 ➔ no data in the mailbox

  – otherwise, count = number of incoming 32-bit words in the mailbox

- Get mailbox data:

```
unsigned int data;
data = spe_read_out_inbox(spe_id);
```

  – data contains next 32-bit word from mailbox

  – routine is non-blocking

  – routine returns MFC_ERROR (0xFFFFFFFF) if no data in mailbox

Write: PPE Mailbox Queues – SPU Calls (spu_mfcio.h)

- “SPU outbound” mailbox
- Check mailbox status:

```
unsigned int count;
count = spu_stat_out_mbox();
```

  – count = 0 ➔ mailbox is full

  – otherwise, count = number of available 32-bit entries in the mailbox

- Put mailbox data:

```
unsigned int data;
spu_write_out_mbox(data);
```

  – data written to mailbox

  – routine blocks if mailbox contains unread data

PPE Interrupting Mailbox Queue – PPE Calls

- "SPU outbound" interrupting mailbox
- Check mailbox status:

  unsigned int count;

  count = spe_stat_out_intr_mbox(spe_id);

  – count = 0 ➔ no data in the mailbox

  – otherwise, count = number of incoming 32-bit words in the mailbox
- Get mailbox data:

  – interrupting mailbox is a privileged register

  – user PPE applications read mailbox data via spe_get_event

PPE Interrupting Mailbox Queues – SPU Calls

- "SPU outbound" interrupting mailbox
- Put mailbox data:

  unsigned int data;

  spe_write_out_intr_mbox(data);

  – data written to interrupting mailbox

  – routine blocks if mailbox contains unread data
- defined in spu_mfcio.h

Write: SPU Mailbox Queue – PPE Calls (libspe.h)

- “SPU inbound” mailbox
- Check mailbox status:

  ```
  unsigned int count;
  count = spe_stat_in_mbox(spe_id);
  ```

  – count = 0 ➔ mailbox is full

  – otherwise, count = number of available 32-bit entries in the mailbox
- Put mailbox data:

  ```
  unsigned int data, result;
  result = spe_write_in_mbox(spe_id,data);
  ```

  – data written to next 32-bit word in mailbox

  – mailbox can overflow

  – routine returns 0xFFFFFFFF on failure

Read: SPU Mailbox Queue – SPU Calls (spu_mfcio.h)

- “SPU inbound” mailbox
- Check mailbox status:

  ```
  unsigned int count;
  count = spu_stat_in_mbox();
  ```

  – count = 0 ➔ no data in the mailbox

  – otherwise, count = number of incoming 32-bit words in the mailbox
- Get mailbox data:

  ```
  unsigned int data;
  data = spu_read_in_mbox();
  ```

  – data contains next 32-bit word from mailbox

  – routine blocks if no data in mailbox

Example using libspe2.x

The PPU program

```
#include <stdio.h>
//#include <libspe.h>
//#include <libmisc.h>
#include <string.h>
#include <libspe2.h>

//spu program
extern spe_program_handle_t getbuf_spu;
//local 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 rc;
```

```
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, NULL,
    &stop_info);
    spe_context_destroy(speid);

    printf("New modified buffer is %s\n", buffer);
    return 0;
}
```

```
DIRS                  = spu
PROGRAM_ppu           = getbuf_dma
IMPORTS  = -lspe2 -lpthread -lmisc \
                        spu/getbuf_spu.a
include $(CELL_TOP)/make.footer
```

The SPU program

```
#include <stdio.h>
#include <string.h>
//#include <libmisc.h>
#include <spu_mfcio.h>
unsigned char buffer[128] __attribute__ ((aligned(128)));
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\"\n", buffer);
        // modify buffer
        strcpy (buffer, “Good Morning!");
        printf("SPE sent to PPU buffer \"%s\"\n", 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;
}
```

```
PROGRAM_spu          := getbuf_spu
LIBRARY_embed   := getbuf_spu.a
IMPORTS              = -lmisc
include $(CELL_TOP)/make.footer
```

DMA Example: Read into Local Store

```
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();
}
```

Read contents of mem_addr into ls_addr

Set tag mask

Wait for all tag DMA completed

Graphics Workloads

Cell Servers for Online Gaming

Motivation
- **Server side physics to enable next generation MMOGs**
- **Current video games perform limited amount of physical simulation**
  - Not enough client CPU resources

Rigid Body Dynamics

- **Objects in the game world are represented by one or more rigid bodies; a sparsely populated world will have about 1000 rigid bodies**
  - 6 degrees of freedom per rigid body
  - Linear position of the body's center of mass and linear velocity are represented by a 3 vector
  - Orientation representation is a unit quaternion
  - Angular velocity is a 3 vector
- **Forces and constraints define interactions between rigid bodies and allow joints, hinges, etc. to be implemented**
- **The physics engine provides real-time simulation of the interaction between the rigid bodies**

Sparse Matrix Data Structures on Cell

- **Matrix is block-sparse with 6x6 blocks**
  - diagonal blocks represent bodies and
  - off-diagonal blocks represent forces between bodies
- **Typical 65-body scene has ~200 nonzero blocks in a 65x65-block matrix**
- **Diagonal elements are assumed nonzero and are stored as a "block" vector for fast access**
- **Off-diagonal elements are stored in linked lists (one per block row) of block data and associated block column position**
- **6x6 float block data is currently stored in column-major form in a padded 8x6 block for ease of access**
- **Vectors used in sparse matrix multiplication are similarly stored with one unused float per three elements**

Numerical Integration

- **Game world is partitioned into non-interacting groups of 1 or more rigid bodies which can be simulated on a single SPU (maximum of about 120 bodies per group).**
- **SPU performs semi-implicit integration step for a second-order rigid body dynamics system using conjugate gradient squared algorithm;**
  - basic operation is multiplication of a 6x6-block-sparse matrix by a vector and multiplication of the matrix transpose by a second vector
- **Output of the integration step gives the change in velocity and angular velocity for each rigid body over one time step**
- **Integration algorithm:**
  1. Calculate the components of A and b. v0 and W are trivial to extract. f0 must be calculated. df_dx and df_dv both require considerable computational effort to calculate.
  2. Form A and b.
  3. solve A*delta_v = b by a conjugate gradient method.
  4. step the system from Y0 to Y1 by delta_v. This is nearly trivial except that integrating orientation is slightly ugly.

SPU Implementation: Rigid Body Structures

```
struct Rigid_Body {
                //state
                Vec3 position;
                Quaternion or Matrix33 orientation;
                Vec3 velocity;
                Vec3 angular_velocity
                //mass params
                float inverse_mass;
                Matrix33 inverse_inertia;
                //other params:
                float coeffecient_friction;
                float coeffecient_damping;
                ...
    } bodies[num_bodies];
The output is logically:
    struct Rigid_Body_Step {
                Vec3 delta_velocity;
                Vec3 delta_angular_velocity;
    } delta_v[num_bodies];
```

The forces can be global, unary, or binary. Here are examples of two common binary forces:

```
struct Point_To_Point_Constraint_Force {
            int index_body_a;
            int index_body_b;
            Vec3 point_body_space_a;
            Vec3 point_body_space_b;
};
struct Contact_Force {
            int index_body_a;
            int index_body_b;
            Vec3 point_world_space;
            Vec3 normal_world_space;
            float penetration;
};
```

Intermediate data structures

- Vec4 v0[2*num_bodies];
- Vec4 f0[2*num_bodies];
- Six component vectors are padded out to 8 components, with each one float of padding on each of the linear and angular components
  - If the SPE calculations were straightforward dense linear algebra, the padding could be dropped, but due to the sparse matrix block granularity, it is better to have the vector components aligned
- The most complicated data structure is the block sparse matrix:

```
struct Block_Sparse_Matrix {
        struct Block {
                Matrix86 m;
                int column_index;
                Element* pointer_next;
        };
        Block* rows[NUM_BODIES];
};
```

- The logically 6x6 blocks are padded to 8x6. The matrix is stored in a column major fashion, with padding on the 4th and 8th element to match padding in v0 and f0:

```
Matrix43  linear_linear,  linear_angular;
Matrix43 angular_linear, angular_angular;
```

- Each row has a singly linked list to the elements. The list is maintained to be sorted by increasing column_index, so that find/insert operations can early out (given that there is never an insert without a find, there is no cost to maintaining this sort order):

```
struct Block_Sparse_Matrix2 {
        struct Block {
                Matrix86 m;
                int column_index;
                Element* pointer_next;
        };
        Block* rows[NUM_BODIES];
};
```

Numerical Integration Steps

Steps 1-4 are performed on the SPE.

1. Calculate the components of **A** and **b**. **v0** and **W** are trivial to extract. **f0** must be calculated. **df_dx** and **df_dv** both require considerable computational effort to calculate.
2. Form **A** and **b**
3. solve **A*delta_v = b** by a conjugate gradient method.
4. step the system from **Y0** to **Y1** by **delta_v**

The steps of the SPE implementation:

1. Initialize **A** and **b** to zero.
2. Construct **A**
   1. By looping over each global, unary, and binary force, and calculating its force contribution and its derivatives, multiplying by the appropriate factors and accumulating into A and b
      1. Example: for a binary force we accumulate **df_dv + h*df_dx** into **A** and **f0 + h*(df_dx*v0)** is accumulated into **b**
      2. For each binary force (between bodies of index **i** and **j**):
         1. Find/allocate the blocks **(i,i), (j,j), (i,j)** and **(j,i)** of **A**

Numerical Integration Steps (cont)

1. Calculate the force - the exact calculation of course depends on what type of binary force is required, but generally uses auxiliary force data (such as body space positions) and the two rigid body's kinematic state.

3. Calculate the derivatives. The force is logically two 6-vectors (one for each body), and its derivative with respect to a 6-vector body state (position or velocity) is logically a 6x6 matrix. A and b are finalized – this involves the h*W premultiply.

A = I - h*w*A

b = h*w*b

4. Solve A=b by a conjugate gradient method.

Why was conjugate gradient squared chosen?

- The preferred choice is bi-conjugate gradient, but this requires multiplies by A transpose
- The sparse matrix transpose times vector can be written in a row-oriented fashion, but having the inner 6x6 logical block efficiently support both multiplication with a logical 6-vector and multiplication of its transpose with a logical 6-vector may be more expensive than the alternative – conjugate gradient squared.
- Caching the transpose of the blocks would likely take too much memory

Conjugate Gradient Squared Method

- **The conjugate gradient squared method only requires A times a vector – however, it has been found in practice to converge more slowly.**
- **Each iteration of the conjugate gradient performs two matrix vector products along with a handful of vector scales, adds, and inner products. The matrix product is the only non-trivial operation. It looks like this:**

```
void mul(Vec8* res, const Block_Sparse_Matrix2& A, const Vec8* x)
{
    for (int i = 0; i < num_bodies; ++i) {
            Vec8 sum = 0;
            for (Block* b=A.rows[i]; b; b = b->pointer_next)
                    sum += b->m * x[b->column_index];
            res[i] = sum;
    }
}
```

Where , b->m * x[b->column_index] is pseudo code for Column_Major_Matrix86 times Vec8 which is basically trivial SPE code.

SPU Sparse Matrix Multiply Code

```
void mul(vf4 d[], const SPU_Sparse_Matrix_Element* const A[], const vf4 x[])
{
        PROFILER(mul);
        int i;
        for (i=0; i < nv/2; ++i) {
                const SPU_Sparse_Matrix_Element* p = A[i];

                vf3 s0 = vf3_zero;
                vf3 s1 = vf3_zero;

                while (p) {
                        int j = p->j;
                        s0 = spu_add(s0, xform_vf3(&p->a.a[0][0], x[2*j+0]));
                        s0 = spu_add(s0, xform_vf3(&p->a.a[0][1], x[2*j+1]));
                        s1 = spu_add(s1, xform_vf3(&p->a.a[1][0], x[2*j+0]));
                        s1 = spu_add(s1, xform_vf3(&p->a.a[1][1], x[2*j+1]));

                        p = p->Pnext;
                }
                d[2*i+0] = s0;
                d[2*i+1] = s1;
        }
}
```

Memory constraints and workload size

- The number of matrix blocks required is less than num_bodies + 2*num_binary_forces
- A typical 65 rigid body scene had approximately 400 contacts and 200 matrix block elements
- SPU memory usage for integrating this example scene follows:

Input:

num_bodies*sizeof(Padded(Rigid_Body)) = 65*160B = 10400B
num_contacts*sizeof(Padded(Contact_Force)) = 400*48B = 19200B
TOTAL= 29600B

Output:

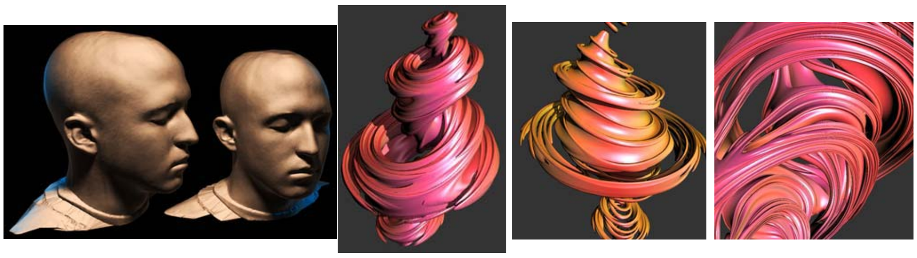
num_bodies*sizeof(Padded(Rigid_Body_Step)) = 65*32B = **2080B**

Intermediate:

num_bodies*sizeof(Padded(W_Element)) = 65*64B = 4160B
num_vectors*num_bodies*sizeof(Padded(Vec6)) = 8*65*32B = 16640B
num_bodies*sizeof(Block*) = 65*4B = 260B
num_blocks*sizeof(Padded(Block)) = 200*208B = 41600B
TOTAL = 62660B

- Including double buffering the input and output areas, we use a **total of 126,020B**
- Maximum workload is probably less than 120 bodies
- Demo

Ray Tracing: Quaternion Julia Sets on the GPU



- Keenan Crane (University of Illinois) – GPU implementation
- Based on "Ray Tracing Deterministic 3-D Fractals" Computer Graphics, Volume 23, Number 3, July 1989
- "This kind of algorithm is pretty much ideal for the GPU - extremely high arithmetic intensity and almost zero bandwidth usage" – Keenan Crane

Optimal Data Organization:
Array of Structures versus Structure of Arrays

(1) Array of Structures

Structure data organization for single triangle

Vertex a	x	y	z	w
Vertex b	x	y	z	w
Vertex c	x	y	z	w

```
Typedef struct _Triangle {
        vector float a, b, c
} Triangles;

Triangles triangles[];
```

- AOS data-packing approach can produce small code sizes, but
  - Typically less than optimal for SIMD architectures
  - Generally requires significant loop-unrolling to improve its efficiency
  - Memory wasted
    - If the vertices contain fewer components than the SIMD vector can hold , e.g., 3 components instead of four

Optimal Data Organization:
Array of Structures versus Structure of Arrays

(2) Structure of Arrays for 4 Triangles

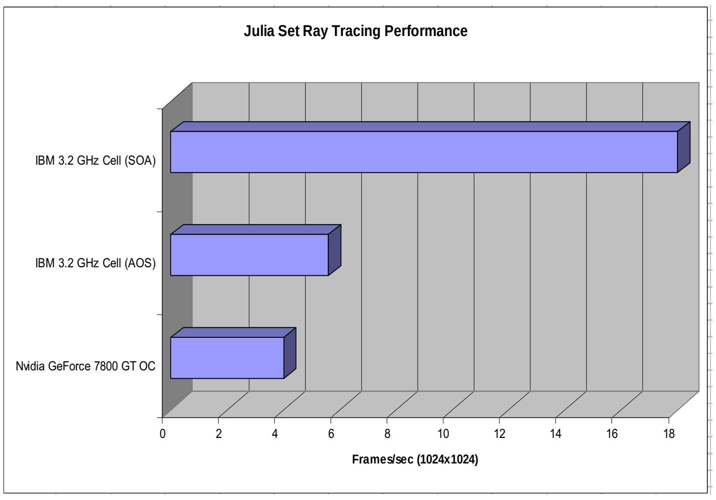
Structure data organization for 4 triangles

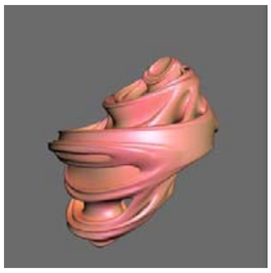
a[0]: x1,x2,x3,x4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
a[1]: y1,y2,y3,y4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
a[2]: z1,z2,z3,z4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
b[0]: x1,x2,x3,x4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
b[1]: y1,y2,y3,y4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
b[2]: z1,z2,z3,z4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
c[0]: x1,x2,x3,x4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
c[1]: y1,y2,y3,y4	Triangle 1	Triangle 2	Triangle 3	Triangle 4
c[2]: z1,z2,z3,z4	Triangle 1	Triangle 2	Triangle 3	Triangle 4

```
Struct Triangles {
    Vector float a[3], b[3], c[3];
}
```

- SOA data-packing approach can be more efficient for some algorithms
  - Typically executes well on SIMD architectures
  - Less memory wasted
  - Usually more complex code

Performance





7 SPEs used for rendering + 1 SPE reserved for image compression

Texture Mapping the Julia Set

- Texture references:
  - Difficult to set up *(predict)* DMAs in advance
  - Significant spatial & temporal locality
  - Small working set size (16-32 kb)
- Texture memory organization
  - Consistency with framebuffer rendering order
    - Tiled framebuffer memory ➔ Tiled texture memory
- Cache layout organization
    - Use cache line size == texture tile size

* Findings from ***The Design and Analysis of a Cache Architecture for Texture Mapping***, Ziyad S. Hakura, and Annop Gupta [Stanford, 1997]

High Level API's

- Simplify programming
  - Hide details of DMA
- Common Operations
  - Cached data read, write
  - Pre-touch
  - Flush
  - Invalidate
  - etc.

```
#include <spe_cache.h>

#define LOAD1(addr)                              \
    * ((char *) spe_cache_rd(addr))

#define STORE1(addr, c)                          \
    * ((char *) spe_cache_wr(addr)) = c


void memcpy_ea(uint dst, uint src, uint size)
{
    while (size > 0) {
        char c = LOAD1(src);
        STORE1(dst, c);
        size--;
        src++;
        dst++;
    }
}
```

Low level Cache API

- Depend on cache type
- Programmer directly controls
  - Look up
  - Branch to miss handler
  - Wait for DMA completion
- Custom interfaces
  - Multiple lookups
  - Special data types
  - Cache locking

```
#include <spe_cache.h>

unsigned int __spe_cache_rd(unsigned int ea) {
        unsigned int ea_aligned = (ea) & ~SPE_CACHELINE_MASK;
        int set, line, byte, missing;
        unsigned int ret;

        missing = _spe_cache_dmap_lookup_(ea_aligned, set);
        line = _spe_cacheline_num_(set);
        byte = _spe_cacheline_byte_offset_(ea);
        ret = *((unsigned int *) &spe_cache_mem[line + byte]);
        if (unlikely(missing)) {
            _spe_cache_miss_(ea_aligned, set, 0, 1);
            spu_writech(22, SPE_CACHE_SET_TAGMASK(set));
            spu_mfcstat(MFC_TAG_UPDATE_ALL);
            ret = *((unsigned int *) &spe_cache_mem[line + byte]);
        }
        return ret;
}
```

Example: SPE Texture Mapping

- *Texturing* **maps images onto 3-D surfaces**
- *Cube environment mapping* **reflects image data from 1 of 6 surrounding texture maps**
- *Fresnel reflection & refraction* **increase realism, complexity of texture look up**
- *Animated 3-D Julia Set Fractal*

Interactive Ray-tracing

Renewed interest from Graphics Community

- Global Illumination
- Rendering time scales sub linearly with scene complexity
- Scales well on multi-core processors
- Mathematically elegant
- Algorithmically simple

Courtesy of Barry Minor, IBM Quasar Design Center

IBM iRT
Interactive Ray-tracer

- **Visualization of Huge Digital Models**
- **Powered by IBM QS20 Blades**
- **720p and 1080p HDTV Output**
- **Seamless Scale Out**
  - **More Blades**
  - **More Cells**
  - **More performance**
- **Real-time Ambient Occlusion**
- **Server Side Rendering**
  - **Image Encode**
  - **IB or Network Image Delivery**
- **Dynamic Load Balancing**
  - **Across Multiple Blades, Cells, & SPEs**

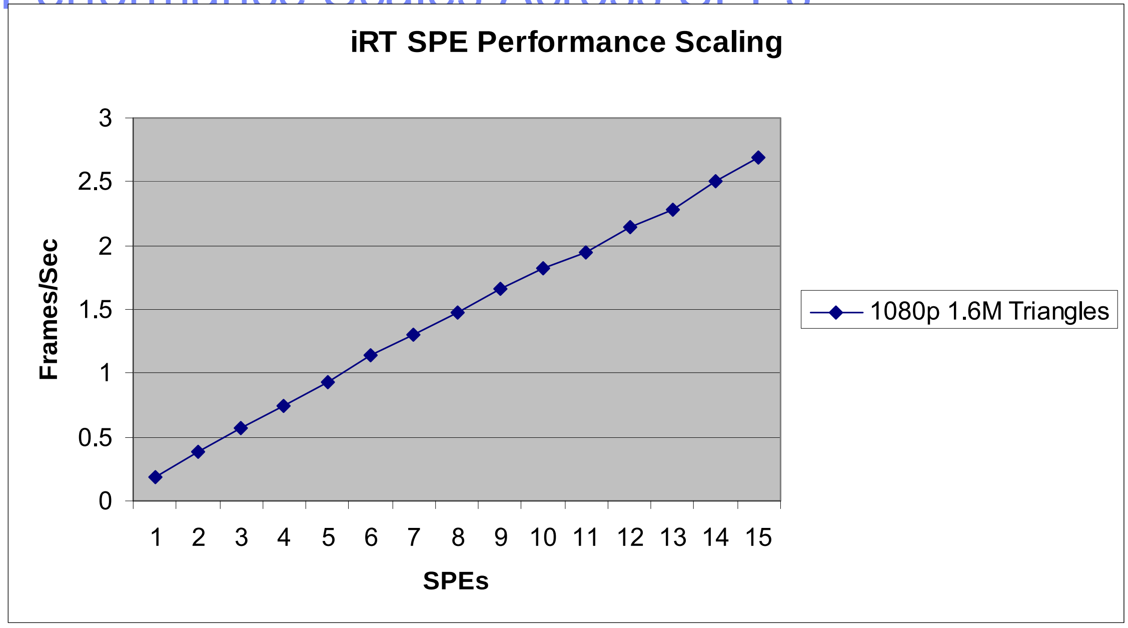
Courtesy of Barry Minor, IBM

Courtesy of Barry Minor, IBM

IBM iRT
Supported Rendering Features

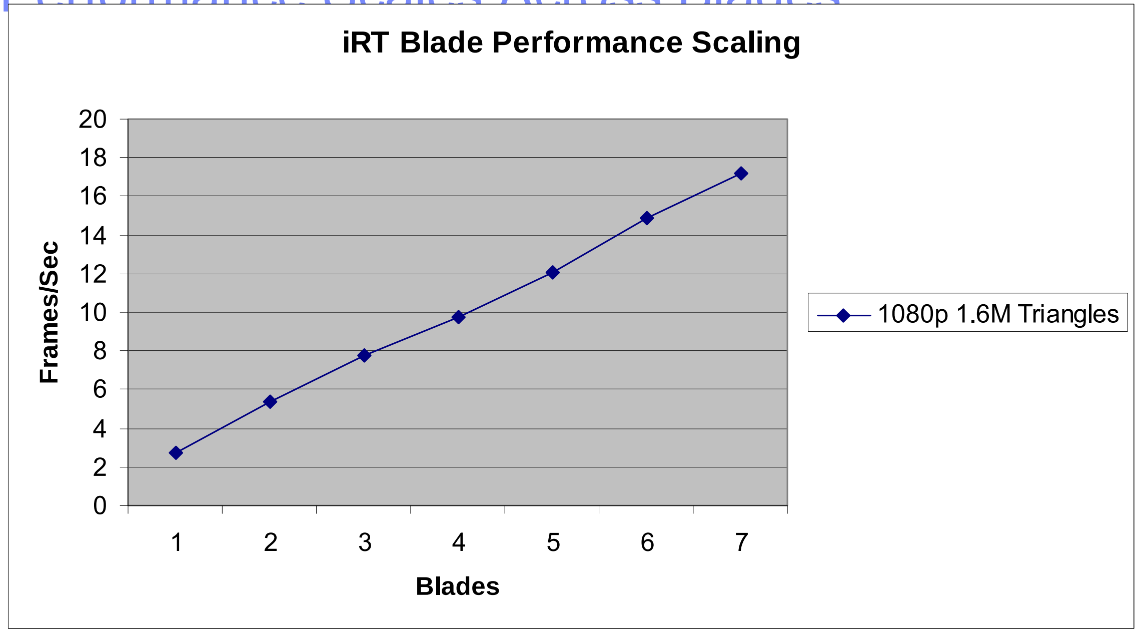
- **Texture Maps**
  - Bilinear Filtering
- **Bump Maps**
  - Blinn Style
- **Phong Lighting Model**
  - Phong Shading
- **Multi-Sampling**
  - 1, 4, 16 Samples per Pixel
  - Jitter Sampled
- **Ambient Occlusion**
  - 4, 16, 64 Random Samples per Primary
- **Optical Effects**
  - Reflection, Refraction

Performance Scales Across SPEs



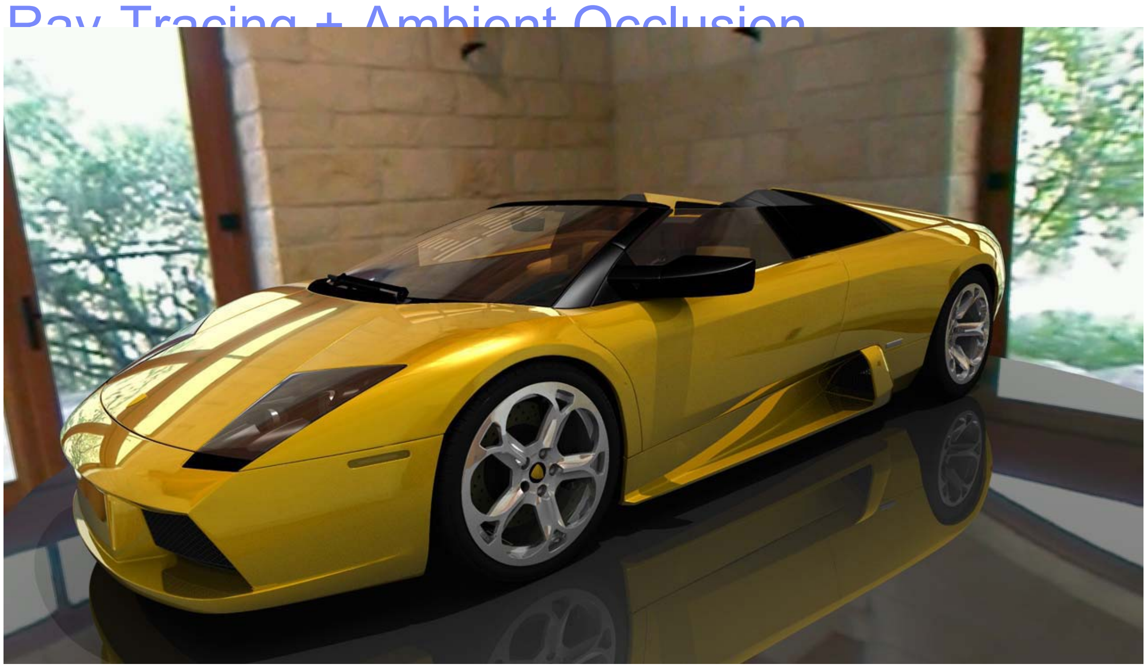
QS20 Blades, FC5, Cell SDK 2.0

Performance Scales Across Blades



QS20 Blades, FC5, Cell SDK 2.0

Ray Tracing + Ambient Occlusion



Primary, Shadow, Secondary, Global illumination – 288 Rays per Pixel

Ray-Triangle Intersection

```
static inline int isect_ray4_triangle (const struct ray4 *ray,
                                        const float4 p[3], hit_rec4 * hit, uint id)
{
  vec_uint4 vid = spu_splats (id);
  vec_float4 p0 = p[0].v;
  vec_float4 p1 = p[1].v;
  vec_float4 p2 = p[2].v;
  vec_float4 ro_x = ray->o.x;
  vec_float4 ro_y = ray->o.y;
  vec_float4 ro_z = ray->o.z;
  vec_float4 rd_x = ray->d.x;
  vec_float4 rd_y = ray->d.y;
  vec_float4 rd_z = ray->d.z;
  vec_float4 edge1 = spu_sub (p1, p0);
  vec_float4 edge2 = spu_sub (p2, p0);
  vec_float4 hit_t = hit->t;
  vec_float4 hit_u = hit->u;
  vec_float4 hit_v = hit->v;
  vec_uint4 hit_id = hit->id;
  vec_float4 one = spu_splats (1.0f);
  vec_float4 zero = spu_splats (0.0f);
  vec_float4 p0_x = spu_splats (spu_extract (p0, 0));
  vec_float4 p0_y = spu_splats (spu_extract (p0, 1));
  vec_float4 p0_z = spu_splats (spu_extract (p0, 2));
  vec_float4 edge1_x = spu_splats (spu_extract (edge1, 0));
  vec_float4 edge1_y = spu_splats (spu_extract (edge1, 1));
  vec_float4 edge1_z = spu_splats (spu_extract (edge1, 2));
  vec_float4 edge2_x = spu_splats (spu_extract (edge2, 0));
  vec_float4 edge2_y = spu_splats (spu_extract (edge2, 1));
  vec_float4 edge2_z = spu_splats (spu_extract (edge2, 2));
```

```
 vec_float4 pvec_x, pvec_y, pvec_z;
  vec_float4 tvec_x, tvec_y, tvec_z;
  vec_float4 qvec_x, qvec_y, qvec_z;
  vec_float4 u, v, t;
  vec_float4 det, inv_det;
  vec_uint4 u_geq_0, v_geq_0;
  vec_uint4 uv_leq_1, t_lt_hit;
  vec_uint4 t_geq_0, valid_hit;

  _CROSS3_V (pvec, rd, edge2);
  det = _DOT3_V (edge1, pvec);
  _INVERSE (inv_det, det);
  _SUB3_V (tvec, ro, p0);
  _CROSS3_V (qvec, tvec, edge1);
  u = spu_mul (_DOT3_V (tvec, pvec), inv_det);
  v = spu_mul (_DOT3_V (rd, qvec), inv_det);
  t = spu_mul (_DOT3_V (edge2, qvec), inv_det);
  u_geq_0 = spu_cmpge (u, zero);
  v_geq_0 = spu_cmpge (v, zero);
  uv_leq_1 = spu_cmple (spu_add (u, v), one);
  t_lt_hit = spu_cmplt (t, hit_t);
  t_geq_0 = spu_cmpge (t, zero);
  valid_hit = spu_and (spu_and (spu_and (u_geq_0, v_geq_0),
                                    spu_and(uv_leq_1, t_lt_hit)), t_geq_0);

  hit->t = spu_sel (hit_t, t, valid_hit);
  hit->u = spu_sel (hit_u, u, valid_hit);
  hit->v = spu_sel (hit_v, v, valid_hit);
  hit->id = spu_sel (hit_id, vid, valid_hit);

  return _any4 (valid_hit) ? 1 : 0;
}
```

Demos

City

Lamborghini

Thank you

Questions?

The RapidMind Development Platform

Michael McCool

Outline

- Basic Concepts
  - Background
  - Performance
  - Architecture
  - Basic vocabulary
  - Defining program objects
  - Parallel programming model
  - Loop conversion example
- Advanced Topics
  - Accessors and copying semantics
  - Applications of dynamic code generation
  - Design patterns
  - Acceleration strategies
  - Program manipulation
- Application Examples
  - Crowd simulation, FFT and convolution, raytracing

- RapidMind Development Platform
  - Single-source solution for portable parallel programming
  - Safe and deterministic data-parallel programming model
  - Scalable to arbitrary number of cores
  - Integrates with existing C++ compilers
- Can be used for programming multiple targets
  - *Unified programming model for both accelerators and CPUs*
  - Support for both GPUs and Cell BE generally available
  - Prototype backend demonstrated on multi-core CPU

- Programmability
  - Just an ISO standard C++ library
  - No new tools or workflow
  - No need for low-level understanding of the processor(s)
  - Expressive, ***safe***, modular, and easy to learn
- Performance
  - Leverages all available computational resources
  - Encourages and supports scalable data parallelism
- Portability
  - Application programming independent of OS or target platform
  - New processors supported without change to application

Programmability

- Use ***existing*** ISO standard C++ compiler:
  - Just include a header file, link to a library
  - Single-source solution, can be used with existing code bases
  - Does not require modification of debugging and build environments
- Allows specification of ***arbitrary computation***:
  - ***NOT*** just a library of canned functions
  - Uses its own runtime optimizing code generator
  - User can specify arbitrary computational kernels
  - Staged compilation strategy avoids overhead of C++

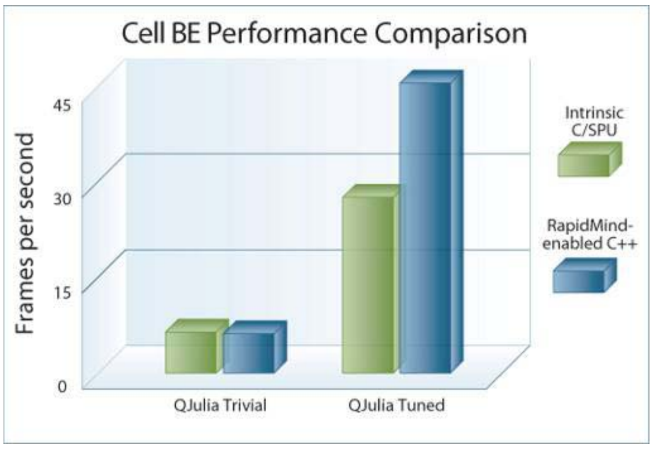
RAPIDMIND

Portability

- Multiple hardware targets:
  - NVIDIA GPUs
  - AMD/ATI GPUs
  - Cell BE
  - Prototype for x86 multi-core demonstrated
- Independent of number of cores
- Independent of memory model
  - Shared or distributed
- If main processor does not change, can support new co-processor ***without*** even recompiling program

RAPIDMIND

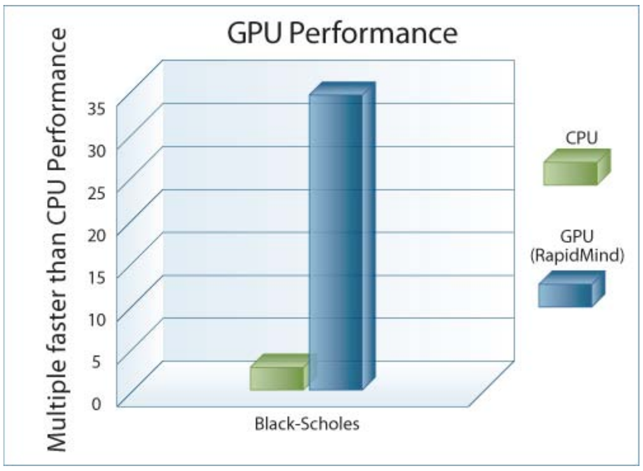
Cell BE Performance



- QJulia application
- Compared with IBM SDK implementation
- Comparable performance with same optimizations
- Additional optimizations possible with only a few lines of code that nearly doubled performance over IBM implementation
- Overall code size and complexity significantly lower than that of IBM SDK implementation

RAPIDMIND

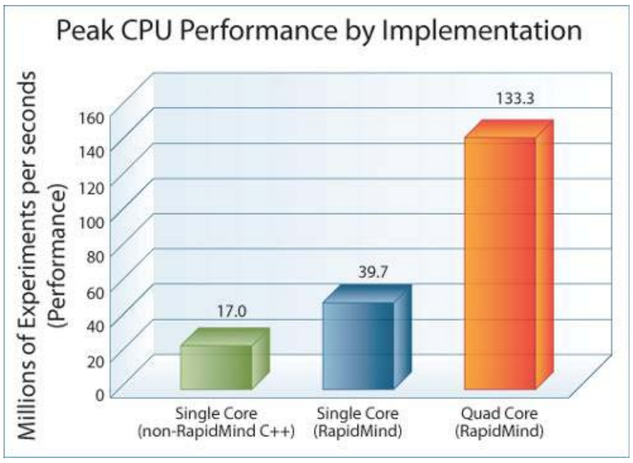
GPU Performance



- Financial quasi Monte-Carlo option-pricing benchmark done in “competition” with HP
- CPU code independently tuned by HP
- GPU implementation over 30x faster than single-core CPU implementation

RAPIDMIND

CPU Performance



- Same financial quasi Monte-Carlo option-pricing benchmark as for GPU benchmark
- RapidMind implementation basically the same as the GPU implementation
- Prototype backend targeting four CPU cores
- RapidMind over 2x faster on one core, 8x faster on four cores

Key Concepts

- ***Vocabulary*** for parallel programming
  - Set of nouns (types) and verbs (operations)
  - *Added* to existing standard language: ISO C++
- A *language* implemented as an *API*

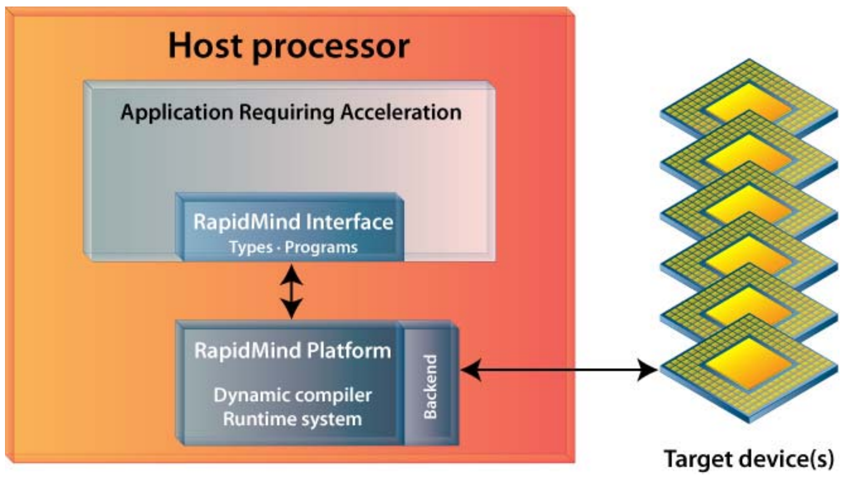
API == Language

- API
  - Issue a sequence of function calls
  - Manipulate state
  - Must issue calls in a certain order
  - Store sequences of calls in buffers (display lists)
  - Play back sequences of calls
- Languages
  - Issue a sequence of statements
  - Manipulate variables
  - Must have a certain syntax
  - Encapsulate sequences of statements in functions
  - Call functions to execute code

RapidMind Platform Interface

- A C++ API
  - for specifying data-parallel computation
- A data-parallel programming language
  - embedded inside C++

RapidMind Platform Architecture



RAPIDMIND

RapidMind Interface

Simple API:
- **Data Types:** Arrays and Values
- **Program Objects:** similar to C++ functions
- **Operations:** C++ and matrix-vector library
- **Collectives:** reductions, scatter, gather, etc.

To use:
- `#include <rapidmind/platform.hpp>`
- `using namespace rapidmind;`
- link to `rmplatform`

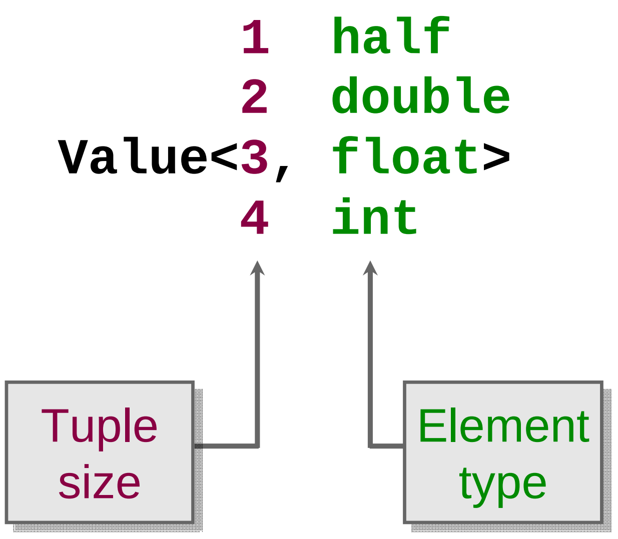
14

RAPIDMIND

Nouns: Basic Types

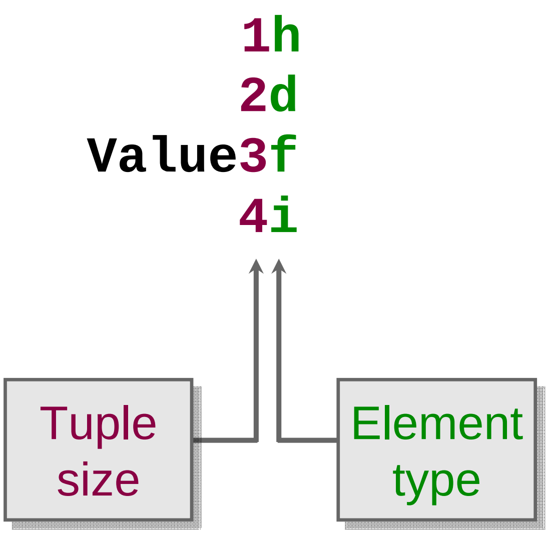
Purpose	Type
Container for fixed-length data	`Value`
Container for variable-sized multidimensional data	`Array`
Container for computations	`Program`

Values



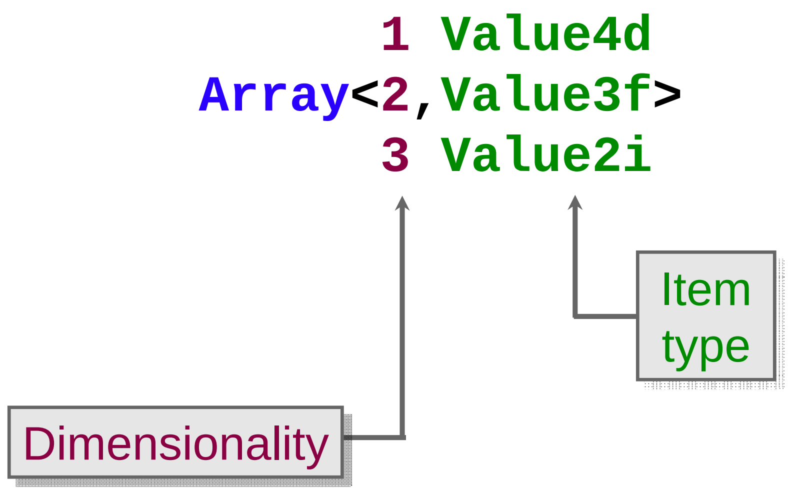
```
      1  half
      2  double
Value<3, float>
      4  int
```

Values



```
     1h
     2d
Value3f
     4i
```

Arrays



```
      1 Value4d
Array<2,Value3f>
      3 Value2i
```

Verbs: Operators

- Operators act componentwise:

  `+, -, *, /, %, &, |, ^, ~, <, …`

- Swizzling and writemasking:

```
Value4f c;
c(2,1,0)
c(0,0,0)
c(1,1,2,3)
c[3]
```

Verbs: Functions

- Can declare functions in the usual way:

```
Value3f
reflect (Value3f v, Value3f n) {
    return Value3f(2.0*dot(n,v)*n - v);
}
```

- Standard library
  - Matrix operations
  - Geometric operations
  - Trigonometry
  - Exponentials and logarithms
  - Splines, interpolation, and polynomials
  - etc.

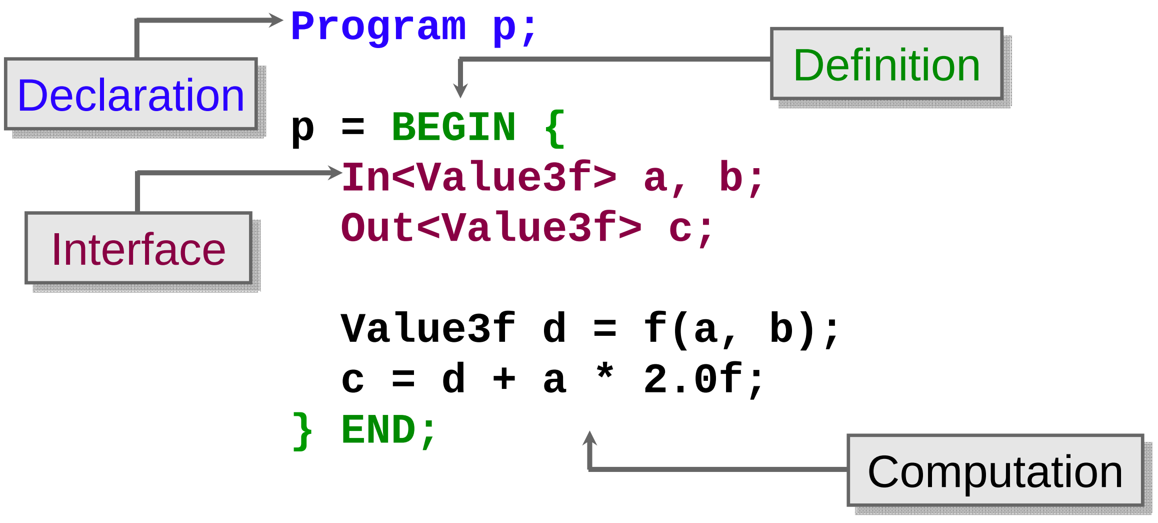
Programs

- ***Immediate mode:***
  - ***Execute*** operations on RapidMind types on host
  - Acts like a standard matrix-vector library
- ***Retained mode:***
  - Enter retained mode with **`BEGIN`**, exit with **`END`**
  - ***Record*** operations on RapidMind types
    - Same operations that work in immediate mode
  - Store operations in **`Program`** object
  - Compile captured operations for coprocessor
    - Dynamic compilation

Dynamic construction of remote procedure call

RAPIDMIND

Program Definition



```
Program p;

p = BEGIN {
    In<Value3f> a, b;
    Out<Value3f> c;

    Value3f d = f(a, b);
    c = d + a * 2.0f;
} END;
```

RAPIDMIND

Program Application

- Apply programs to arrays, get new arrays

```
C = p(A,B);
```

Invokes parallel execution

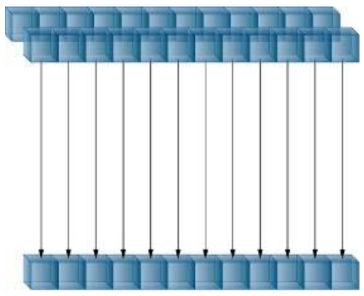
Array Semantics

- Arrays use by-value semantics
  - Can assign arrays with O(1) cost
  - Strong modularity
  - Simple and easy to understand
  - Consistent with value tuples
- Most data copies can be optimized away
  - Copies only required to complete partial updates
  - Parallel assignment means partial updates can be avoided
- By-reference semantics available:
  - Via the **`ArrayAccessor`** type

SPMD Data Parallel Programming Model

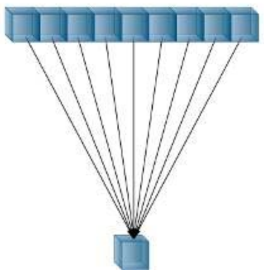
Apply functions to arrays:

- Application: **`C = f(A,B)`**
- May have control flow (SPMD model)
- May perform random reads from other arrays
- Can read and write to subarrays



Apply collective operations to arrays:

- Reduce: **`a = reduce(p,A)`**
- Gather: **`A = B[U]`**
- Scatter: **`A[U] = B`**
- *Others…*



Control Flow

```
Program p;

p = BEGIN {
  In<Value3f> a, b;
  Out<Value3f> c;

  Value3f d = f(a, b);
  IF (all(a > 0.0f)) {
    c = d + a * 2.0f;
  } ELSE {
    c = d – a * 2.0f;
  } ENDIF;
} END;
```

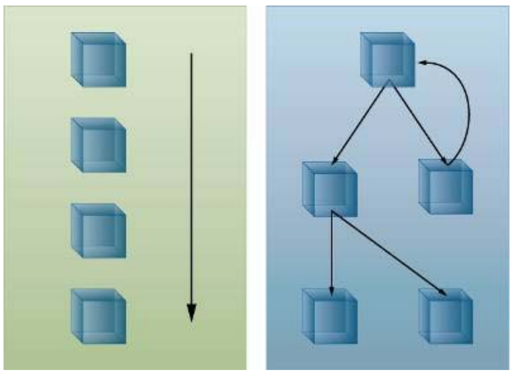
Control Flow: SPMD vs. SIMD

SIMD:

- ***Single Instruction, Multiple Data***
- Kernels include sequences of simple instructions
- Take constant amount of time to execute

SPMD:

- ***Single Program, Multiple Data***
- Kernels may include control flow (loops and conditionals)
- Can avoid unnecessary work



SPMD includes but is *intrinsically* more powerful than SIMD

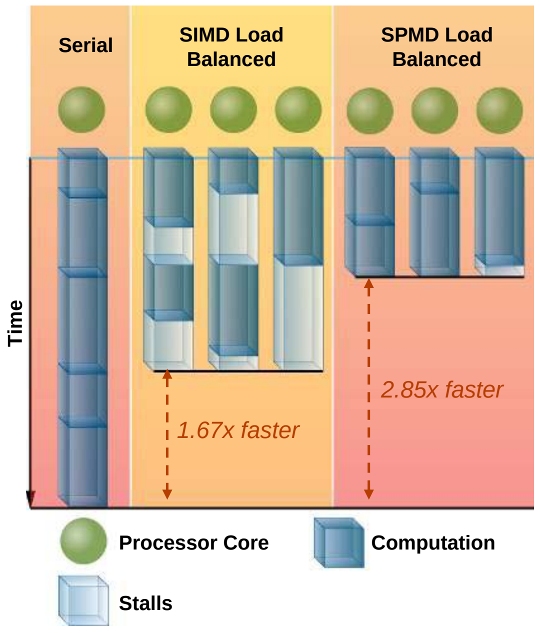
Load Balancing

SIMD scheduling

- Assumes constant time per kernel

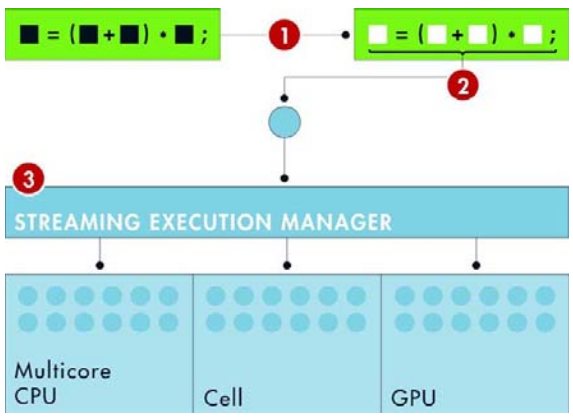
SPMD scheduling

- Takes variable execution time into account
- Load balancing distributes workload evenly across cores



Conversion Example

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```



RAPIDMIND

0. Access API

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;
```

RAPIDMIND

1. Replace Types

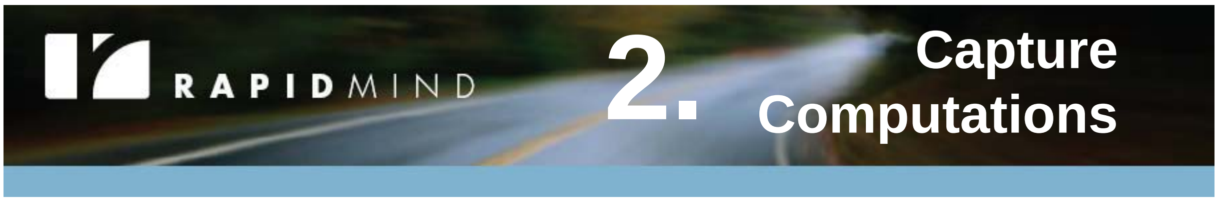
```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}
```

1b. Replace Types

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

template <typename T>
T func(
 T r, T s
) {
 return (r + s) * f;
}
```



2. Capture Computations

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 . . .
}
```

RAPIDMIND

3. Parallel Execution

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++)
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 a = func_prog(a,b);
}
```

RAPIDMIND

Usage Summary

- Usage:
  - Include platform header
  - Link to runtime library
- Data:
  - Tuples
  - Arrays
  - *Remote data abstraction*
- Programs:
  - Defined dynamically
  - Execute on coprocessors
  - *Remote procedure abstraction*

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 a = func_prog(a,b);
}
```

Feature Summary

- Abstractions for *both* code *and* data
- Generate and manipulate code explicitly
  - C++ modularity
  - FORTRAN execution efficiency
- Can target GPU as well as Cell BE
- Simple, safe programming model
- Single-source ISO standard C++ program:
  - ***No extensions needed***
  - ***Use your existing compiler***

Advanced Topics

- Accessors
  - Extracting and accessing subarrays
  - Copying semantics
- Metaprogramming
  - Applications of dynamic code generation
- Design patterns
  - Processor pattern
  - Compiler pattern
- Acceleration strategies
  - Loop conversion
  - Interpreter conversion
  - Task conversion
- Program manipulation
  - Program algebra

Accessors

`offset(A,n)`
- Drop first n elements of A

`shift(A,n)`
- Translate index into array A by n

`take(A,n)`
- Drop all but first n elements of A

`slice(A,i,j)`
- Extract subarray from i to j, inclusive

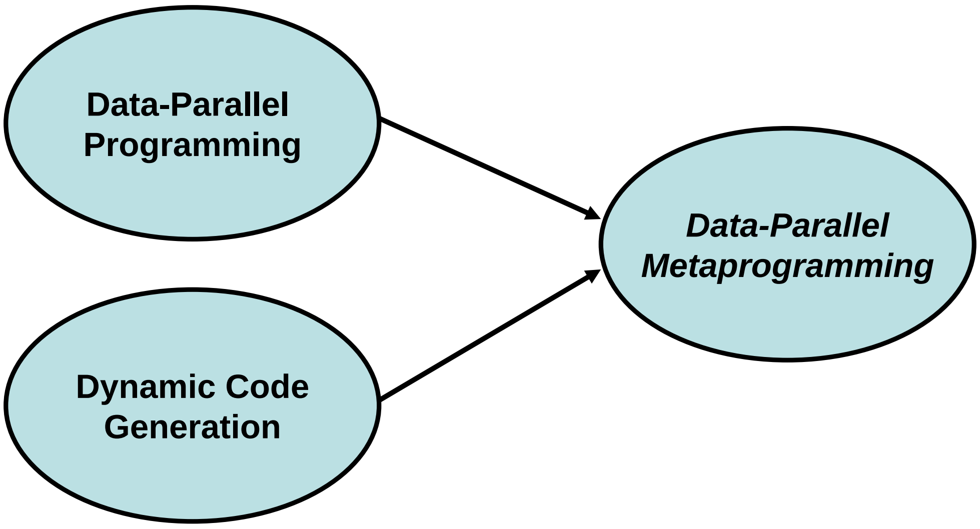
`stride(A,k)`
- Extract every kth element

Return instance of **`ArrayAccessor`** type
- *References* subarray “view”, does not copy

Copying Semantics

- Assignment to an **`Array`**:
  - by-value
  - assignment *replaces* destination
  - allocates new memory if needed
- Assignment to an **`ArrayAccessor`**:
  - by-value
  - assignment *copies* into destination
- Explicit copying can be forced with **`copy`** function
- Memory automatically freed if no longer referenced

Metaprogramming: Dynamic Code Generation



Advantages of Data Parallelism

- Efficient on a variety of computer architectures
  – Shared memory machines
  – Distributed memory machines
  – Vector/stream machines
- Predictable memory access patterns
- Scales to arbitrary number of processors
- Single thread of control
  – Simple extension of existing programming practice
  – No explicit synchronization needed
  – No deadlocks or non-determinism
  – Debugging simplified

Advantages of Metaprogramming

- Object-oriented overhead of C++ avoided
  - Platform *only* compiles operations on RapidMind types
  - ***Structure with C++:*** templates, objects, namespaces, …
  - ***Run like FORTRAN (or better)***
- Metaprogramming can be used to build
  - Parameterized code, with possible automatic tuning
  - Code generated algorithmically
  - Code that adapts to hardware platform
  - Code that adapts to or is generated based on data
  - Compilers from interpreters
  - Higher order functions to parameterize operations

Design Patterns

- Processor pattern
  - Manage code generation and initialization
  - Encapsulate parameterized code
- Compiler pattern
  - Remove overhead from computation specified at runtime

Processor Pattern

```
template <typename T, typename S>
class Processor {
 protected:
   S m_f;

   T m_func(
     T r, T s
   ) {
     return (r + s) * m_f;
   }

   Program m_prog;

 public:
   Processor(
     S f
   ): m_f(f) {
     m_prog = BEGIN {
       In<T> r, s;
       Out<T> q;
       q = m_func(r,s);
     } END;
   }

   Array<2,T>
   apply(
     const Array<2,T>& a,
     const Array<2,T>& b
   ) {
     return m_prog(a,b);
   }
};
```

```
// USAGE

// Initialize
Value1f g;
Processor<Value3f,Value1f> proc(g);

// Apply
Array<2,Value3f> p(512,512);
Array<2,Value3f> q(512,512);
p = proc.apply(p,q);
```

Compiler Pattern

Problem:

- Need to evaluate some expression not known until runtime
- Example:
  - Image compositing
  - User may express sequence of operations in visual language

Solution 1: Interpreter Pattern

1. Encode computation in data structure (ex: operator dag)
2. Traverse data structure, *executing* operations
3. Return result

Solution 2: Compiler Pattern

1. Encode computation in data structure (ex: operator dag)
2. Traverse data structure, *recording* operations
3. Compile operations into program object
4. Execute program object on data
5. Return result

Approach 1: Loop Conversion

- Find hot spot
- Identify loop structures
- Convert loops to parallel operations

Approach 2: Interpreter Conversion

- Identify use of interpreter pattern
- Convert to compiler pattern

Advantages:

- Can collect a significant amount of computation together even when there is no obvious hot spot
- Can avoid memory and branching overhead of interpretation

RAPIDMIND

Accelerating Applications

Approach 3: Task Conversion

- Identify use or potential for task parallelism
- Convert to SPMD model
- Use arrays to communicate between tasks

Advantages:

- Simplified debugging
- Bulk synchronous model

RAPIDMIND

Program Manipulation

- Combination:
  - Program “algebra” to combine programs into new programs
  - Can use to modify interfaces to existing programs
  - Can use to specialize existing programs
- Partial evaluation:
  - Can bind inputs one at a time
  - Can convert inputs to non-local variables and vice versa
- Introspection:
  - Can analyze program interface and performance at runtime
  - Use for self-tuning libraries

Program Algebra

- **Algebra:**
  - Set of objects
  - Set of operators
  - Closed
- *Objects:*
  - Programs
- *Operators:*
  - Functional composition:
    ```
    p << q
    ```
  - Concatenation:
    ```
    bundle(p,q)
    ```

Applications of the Program Algebra

- Interface adaptation
  - Reordering
  - Packing/unpacking
  - Input or output type conversion
- Specialization
  - Discard unneeded outputs
  - Eliminates unnecessary computation
- Pipelining
  - Combine producer/consumer programs into one:
    ```
    A = (p << q << r)(B);
    ```
  - Implement pipeline as single data-parallel task

Partial Evaluation

- Can bind only some inputs of a program, not all
- Binding gives a new program with fewer inputs
  - If bind only 1 input of an *n* input program
  - Get back program with *n*-1 inputs
- Partial evaluation provides
  - Flexibility
  - Interface adaptation
  - Optimization opportunities
- Two kinds of binding:
  - Tight: uses **`()`**
  - Loose: uses **`<<;`** is invertible using **`>>`**

Tight Binding

- Tight binding:

```
Program q = p(A);
```

- Execution can be deferred
- When eventually executes:
  - Uses value of **A** in effect at time of *binding*
  - Compiler can use actual value of **A** to optimize code

RAPIDMIND

Loose Binding

- Loose binding:

```
Program q = p << A;
```

- Execution can be deferred
- When eventually executes:
  - Uses value of **A** in effect at time of *execution*
  - Value of **A** can be used to parameterize execution
- **A** acts like a non-local variable

RAPIDMIND

Unbinding

- Convert input to non-local variable:

```
q = p << A;
```

- Convert non-local variable to input:

```
q = p >> A;
```

Applications

Examples

- *Crowd simulation (GDC)*
- *Ray tracing (w/ RTT)*
- *Fast Fourier transform*
- *Convolution*
- Quasi Monte Carlo option pricing
- Matrix-matrix multiply (SGEMM)
- Transformation and lighting
- Color and gamma correction
- Object tracking
- Sorting
- Quaternion Julia set
- Deferred shading
- Vector textures
- *Others…*

Crowd Simulation

BUILT WITH RAPIDMIND
visit www.rapidmind.net

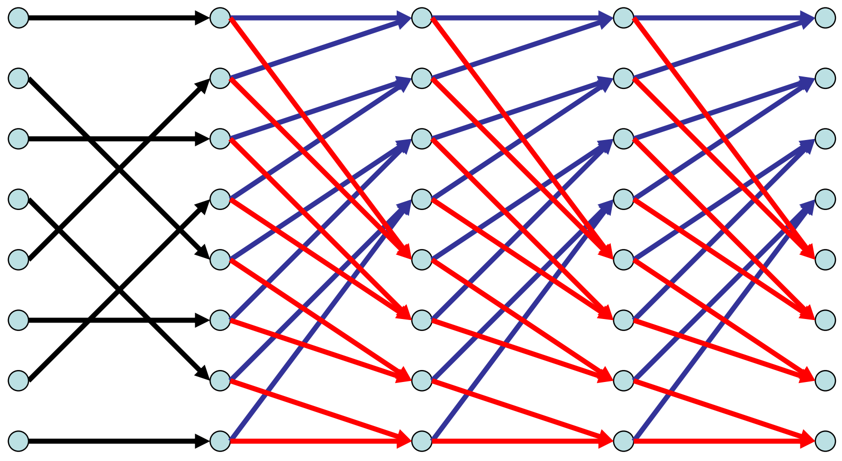
Crowd Simulation

- Graphics on GPU
  - Shaders implemented using RapidMind platform
- Behavioral Simulation on Cell BE Blade
  - 16K autonomous characters (4K visible at once)
- Parallel Execution:
  - Rules to simulate social behavior and basic physics
- Global Communication:
  - Any character can interact with any other
    - Requires (approximate) solution to K-nearest-neighbor problem
  - Behavior depends on the environment
    - Random access to environmental parameter grid
    - Obstacles, ground cover and slope

Fast Fourier Transform

- Fundamental signal processing operation
  - Image processing
  - Pattern matching
  - Solving differential equations
- Standard test case for parallel computation
- Involves both
  - Computation
  - Communication
- Many varieties and ways to implement
  - Will show radix-2 split-stream complex-to-complex 1D FFT

Signal Flow Graph



Fast Fourier Transform

```
// Fast Fourier Transform
Array<1,Value2f>
FFT (Array<1,Value2f> data, int n) {
    int N = (1 << n);

    // define program objects
    …

     // generate and scramble twiddle factors with gather
    …

    // scramble input data using a gather
    …

    // perform split-stream FFT using lg(N) passes
    …
}
```

RAPIDMIND

Fast Fourier Transform

```
// define program objects
Program butterfly_A = BEGIN {
    In<Value2f> a, b;
    Out<Value2f> c = a + b;
} END;

Program butterfly_B = BEGIN {
    In<Value2f> a, b, w;
    Value2f t = a - b;
    Out<Value2f> c;
    c[0] = t[0]*w[0] + t[1]*w[1];
    c[1] = t[1]*w[0] - t[0]*w[1];
} END;
```

RAPIDMIND

Fast Fourier Transform

```
// generate and scramble twiddle factors with gather
Array<1,Value2f> w(N/2);
w = twiddle(n-1)[ bitreverse(n-1) ];

// allocate temporary storage
Array<1,Value2f> x[2];
x[0] = Array<1,Value2f>(N);
x[1] = Array<1,Value2f>(N);

// scramble input data using a gather
x[0] = data[ bitreverse(n) ];

// initialize source marker
int src = 0;
```

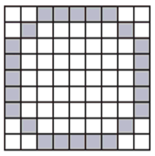
Fast Fourier Transform

```
// perform split-stream FFT using log(N) passes
for (int k=n-1; k>=0; k--) {
    // write into lower half of output array
    take(x[!src],N/2) = butterfly_A(
      stride(x[src],2),
      stride(offset(x[src],1),2)
    );
    // write into upper half of output array
    offset(x[!src],N/2) = butterfly_B(
      stride(x[src],2),
      stride(offset(x[src],1),2),
      take(w,1<<k)
    );
    // swap source and destination buffers
    src = !src;
}
// return final transform
return x[src];
```

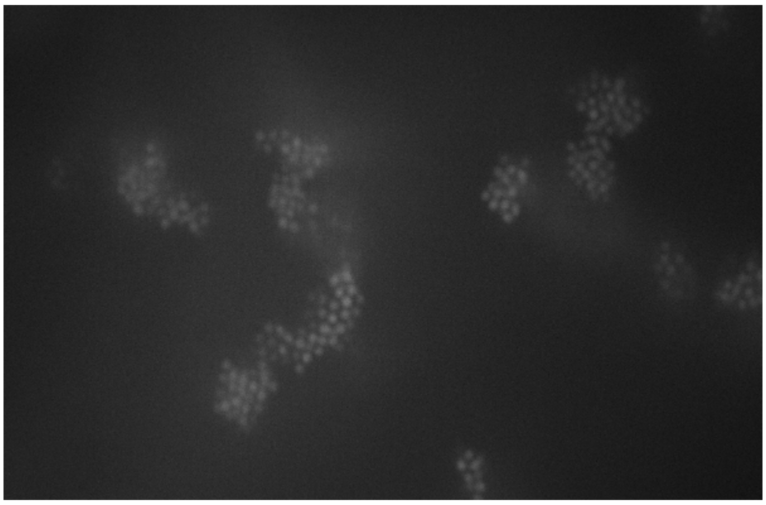
Convolution

- Fundamental signal processing operation
- For large filters, use FFT
  - FFT
  - Elementwise complex multiplication
  - Inverse FFT
- For small filters, do directly
  - Shift flipped filter to each pixel, multiply, sum
  - May process many images with one filter
  - Filters used in pattern matching may be sparse
  - Can exploit sparsity to get more efficient execution

Convolution



*



Confocal microscopy image
courtesy of Peter J. Lu, Harvard

Convolution

```
float filter[N0][N1];
Array<2,Value1f> image(M0,M1);

Program convolve = BEGIN {
  In<Value2i> u;
  Out<Value1f> result = Value1f(0.0f);
  for (int i = 0; i < N0; i++) {
    for (int j = 0; j < N1; j++) {
      if (filter[i][j] != 0.0f) {
        Value2i tap = u - Value2i(i,j);
        result += filter[i][j] * image[tap];
      }
    }
  }
} END;

image = convolve << grid(M0,M1);
```

Raytracing

- Real-time raytracing
  - Supports reflection and refraction
  - Many recursive rays per pixel
  - Incoherent memory access
  - Accelerator data structure traversal
- Commercial product:
  - Developed by RTT AG, Germany
  - Used for automotive CAD visualization
- Hardware:
  - Released product runs on GPUs
  - Demonstrated on Cell BE at SIGGRAPH