Cell Programming Tutorial

Rodric Rabbah

Acknowledgments:
David Zhang and Phil Sung
6.189 IAP 2007 TAs
Get a PS3, Add Linux

- The PS3 can boot user installed Operating Systems
  - Dual boot: GameOS and Other OS

- Installing Linux on the PS3 is well documented
  - Yellow Dog Linux
  - Fedora Core Linux
  - Other Linux distributions reportedly work as well

- User level access to the PS3 processor: Cell
  - Cell SDK from IBM alphaWorks adds compilers, examples, etc.
Your Projects

- Each of you assigned a PS3 and given login information
- All SDKs and related software installed
- For support contact rabbah@mit.edu
Cell Recap

- **Cell**: 9 cores on single chip
  - 1 PPE (PPU + Cache)
  - 8 SPEs (SPU + Local Store)
- **PPE** is a general-purpose PowerPC processor
  - Runs operating system, controls SPEs
- **SPEs** optimized for data processing
- Cores are connected to each other and memory through high-bandwidth Element Interconnect Bus

Courtesy of International Business Machines Corporation.
Unauthorized use not permitted.
Programming Cell

- **Multiple programs in one**
  - PPU and SPU programs cooperate to carry out computation

- **SPE local store**
  - 256 KB of low-latency storage on each SPE
  - SPU loads/stores/ifetch can only access local store
  - Accesses to main memory done through DMA engine
  - Allows program to control and schedule memory accesses
  - Something new to worry about, but potential to be much more efficient

- **SIMD**
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
A Simple Cell Program

**PPU (hello.c)**

```c
#include <stdio.h>
#include <libspe.h>

extern spe_program_handle_t hello_spu;

int main() {
    speid_t id[8];
    // Create 8 SPU threads
    for (int i = 0; i < 8; i++) {
        id[i] = spe_create_thread(0,
                                  &hello_spu,
                                  NULL,
                                  NULL,
                                  -1,
                                  0);
    }
    // Wait for all threads to exit
    for (int i = 0; i < 8; i++) {
        spe_wait(id[i], NULL, 0);
    }
    return 0;
}
```

**SPU (hello_spu.c)**

```c
#include <stdio.h>

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

- **spe_create_thread**
  ```c
  speid_t
  spe_create_thread(thread group, program handle, argp, envp, <...>)
  ```

- **spe_wait, spe_kill**

- **spe_read_out_mbox, spe_write_in_mbox, spe_write_signal**

- **spe_get_ls**
  - Returns memory-mapped address of SPU’s local store
  - PPU/other SPUs can DMA using this address

- **spe_get_ps_area**
  - Returns memory-mapped address of SPU’s MMIO registers
    - MMIO: memory mapped I/O (covered later)

- **SPU program**
  ```c
  int
  main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
  ```
SPE Threads

- Not the same as “normal” threads
- SPE does not have protection, can only run one thread at a time
  - PPU can “forcibly” context-switch a SPE by saving context, copying out old local store/context, copying in new
- Current SDK does not support context switching SPEs
  - SPE threads are run on physical SPEs in FIFO order
  - If more threads than SPEs, additional threads will wait for running threads to exit before starting
- Don’t create more threads than physical SPEs
- Cell processor has 8 physical SPEs, 6 enabled on PS3 Linux
Exercise

- Compile and run hello example
  - Fetch tarball
    wget http://cag.csail.mit.edu/ps3/recitation1/examples.tar.gz
  - Unpack tarball
    tar zxf examples.tar.gz
  - Go to hello example
    cd examples/hello
  - Compile SPU program
    cd spu
    /opt/ibmcmp/xlc/8.1/bin/spuxlc -o hello_spu hello_spu.c -g -Wl,-N embedspu -m32 hello_spu hello_spu hello_spu-embed.o
    ar –qcs hello_spu.a hello_spu-embed.o
  - Compile PPU program
    cd ..
    /opt/ibmcmp/xlc/8.1/bin/ppuxlc -o hello hello.c -g -Wl,-m,elf32ppc spu/hello_spu.a -lspe
  - Run
    ./hello
Exercise

- Use make build system for easier compilation process
- Compile using the make build system
  - Set environment variable $CELL_TOP
    export CELL_TOP=/opt.ibm/cell-sdk/prototype
  - Remove previously compiled code in directory
    make clean
  - Rebuild the program
    make
  - Run
    ./hello
Agenda

● Computing with SPEs
● SIMD
● Functional debugging
● Performance debugging
Data In and Out of the SPE

- SPU needs data
  1. SPU initiates DMA request for data
Data In and Out of the SPE

- SPU needs data
  1. SPU initiates DMA request for data
  2. DMA requests data from memory
Data In and Out of the SPE

- SPU needs data
  1. SPU initiates DMA request for data
  2. DMA requests data from memory
  3. Data is **copied** to local store
    - Once copied, data is no longer coherent with the rest of the system
    - Changes to data in local store not reflected in main memory (and vice versa)
Data In and Out of the SPE

- SPU needs data
  1. SPU initiates DMA request for data
  2. DMA requests data from memory
  3. Data is copied to local store
  4. SPU can access data from local store
Data In and Out of the SPE

- SPU needs data
  1. SPU initiates DMA request for data
  2. DMA requests data from memory
  3. Data is copied to local store
  4. SPU can access data from local store
- SPU operates on data then **copies**
  data from local store back to memory
  in a similar process
DMA Example

- From SPU
  
  ```c
  mfc_get(destination LS addr,
           source memory addr,
           # bytes,
           tag,
           <...>)
  
  mfc_put(source LS addr,
           destination memory addr,
           # bytes,
           tag,
           <...>)
  ```

- `mfc_stat_cmd_queue`
  - Queries number of free DMA command slots
  - Similar functions to query available mailbox/signal entries

- From PPU (libspe)
  
  - `spe_mfc_get`, `spe_mfc_put`
DMA Example

- Array of integers in memory that we want to process on SPU
- Need to tell SPU program
  - Location (address) of array
  - Size of array
  - Additional parameters?
- Approach
  - Fill in control block in main memory
  - Pass address of control block to SPU
  - Have SPU DMA control block to local store

```c
typedef struct {
    uintptr32_t data_addr;
    uint32_t num_elements;
    ...
} CONTROL_BLOCK;
```

Generic C code
```
for (int i = 0; i < NUM_ELEMENTS; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```
DMA Example

// Data array
int data[NUM_ELEMENTS] __attribute__((aligned(128)));

CONTROL_BLOCK cb __attribute__((aligned(16)));

int main() {
    ...

    // Fill in control block
    cb.data_addr = data;
    cb.num_elements = NUM_ELEMENTS;

    // Create SPU thread
    id = spe_create_thread(0, &dma_spup, &cb,
        NULL, ...);

    CONTROL_BLOCK cb __attribute__((aligned(16)));
    int *data;

    int main(speid, argp, envp) {
        // DMA over control block
        mfc_get(&cb, argp, sizeof(cb), 5, ...);

        // Mask out tag we're interested in
        mfc_write_tag_mask(1 << 5);

        // Wait for DMA completion
        mfc_read_tag_status_all();

        // Compare mfc_read_tag_status_any/immediate

        // Allocate 128-byte aligned buffer
        data = malloc_align(data_size, 7);

        // DMA over actual data
        mfc_get(data, cb.data_addr, data_size, 5, ...);

        // Wait for DMA completion
        mfc_read_tag_status_all();

        for (int i = 0; i < NUM_ELEMENTS; i++) {
            data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
        }
    }

    for (int i = 0; i < NUM_ELEMENTS; i++) {
        data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
    }

    Generic C code

    for (int i = 0; i < NUM_ELEMENTS; i++) {
        data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
    }
DMA Example

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

PPU

```
// Process the data
for (int i = 0; i < cb.num_elements; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}

// DMA back results
mfc_put(data, cb.data_addr, data_size, 5, ...);

// Wait for DMA completion
mfc_read_tag_status_all();

// Notify PPU using outbound mailbox
spu_write_out_mbox(0);
return 0;
```

SPU

- Assumed entire array fits in one DMA command (16 KB)
- Assumed array size is multiple of 16 bytes

PPU

```
// Wait for mailbox message from SPU
while (spe_stat_out_mbox(id) == 0);

// Drain mailbox
spe_read_out_mbox(id);

// Done!
...
```
DMA Example 2

- Add 2 arrays of integers and store result in 3rd array
- Same approach
  - Fill in control block in main memory
  - Pass address of control block to SPU
  - SPU DMAs control block to LS
  - SPU DMAs both input arrays to LS
  - SPU DMAs result back to memory

Generic C code

```c
for (int i = 0; i < NUM_ELEMENTS; i++) {
    result[i] = data1[i] + data2[i];
}
```

typedef struct {
    uintptr32_t data1_addr;
    uintptr32_t data2_addr;
    uintptr32_t result_addr;
    uint32_t num_elements;
    ... 
} CONTROL_BLOCK;

DMA Example 2

// Data and result arrays
int data1[NUM_ELEMENTS] __attribute__((aligned(128)));
int data2[NUM_ELEMENTS] __attribute__((aligned(128)));
int result[NUM_ELEMENTS] __attribute__((aligned(128)));

CONTROL_BLOCK cb __attribute__((aligned(16)));

int main() {
    ...  
    // Fill in control block
    cb.data1_addr = data1;
    cb.data2_addr = data2;
    cb.result_addr = result;
    cb.num_elements = NUM_ELEMENTS;

    // Create SPU thread
    id = spe_create_thread(0, &dma_spu, &cb, NULL, ...);

    for (int i = 0; i < NUM_ELEMENTS; i++) {  
        result[i] = data1[i] + data2[i];
    }

    ...  
}

for (int i = 0; i < NUM_ELEMENTS; i++) {  
    result[i] = data1[i] + data2[i];
}

PPU

SPU

Generic C code

for (int i = 0; i < NUM_ELEMENTS; i++) {  
    result[i] = data1[i] + data2[i];
}

PPU

SPU

Generic C code
### DMA Example 2

#### Generic C code

```c
for (int i = 0; i < NUM_ELEMENTS; i++) {
    result[i] = data1[i] + data2[i];
}
```

- `// Process the data`
- `// DMA back results`
- `// Wait for DMA completion`
- `// Notify PPU using outbound mailbox`

#### Same assumptions
- Each array fits in one DMA command (16 KB)
- Array sizes are multiples of 16 bytes

---

// Wait for mailbox message from SPU
while (spe_stat_out_mbox(id) == 0);

// Drain mailbox
spe_read_out_mbox(id);

// Done!
...
SPE-SPE DMA Example

- Streaming data from SPE to SPE
- Distribute computation so one SPE does multiplication, another does addition
- Keep actual data transfer local store to local store
- Communication?
  - PPE orchestrates all communication
  - SPEs talk to each other via mailboxes/signals

```c
for (int i = 0; i < cb.num_elements; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```
SPE-SPE DMA Example

- SPEs that communicate with each other need to know:
  - Addresses of local stores
  - Addresses of memory mapped I/O (MMIO) registers
- Only PPU program (via libspe) has access to this information
  - PPU creates SPE threads, gathers address information, informs SPEs
// Create SPE threads for multiplier and adder
id[0] = spe_create_thread(0, &dma_spu0, 0, NULL, ...);
id[1] = spe_create_thread(0, &dma_spu1, 1, NULL, ...);
typedef struct {
        uintptr32_t spu_ls[2];
        uintptr32_t spu_control[2];
        ...
    } CONTROL_BLOCK;

// Fill in control block
for (int i = 0; i < 2; i++) {
    cb.spu_ls[i] = spe_get_ls(id[i]);
    cb.spu_control[i] = spe_get_ps_area(id[i], SPE_CONTROL_AREA);
}
...

// Send control block address to all SPUs
for (int i = 0; i < 2; i++) {
    spe_write_in_mbox(id[i], &cb);
}

// Wait for control block address from PPU
cb_addr = spu_read_in_mbox();

// DMA over control block and wait until done
mfc_get(&cb, cb_addr, sizeof(cb), 5, ...);
mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();

Both SPUs
SPE-SPE DMA Example

// DMA in data from memory and wait until complete
mfc_get(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
SPE-SPE DMA Example

// Process data
...

PPU

Initialization

SPU 0

SPU 1

Data

SPU 0
SPE-SPE DMA Example

---

// Temporary area used to store values to be sent to mailboxes with proper alignment.
struct {
    uint32_t padding[3];
    uint32_t value;
} next_mbox __attribute__((aligned(16)));

// Notify SPU 1 that data is ready. Send over virtual address so SPU 1 can copy it out.
next_mbox.value = cb.spu_ls[0] + data;
mfc_put(&next_mbox.value,
    cb.spu_control[1] + 12,
    4,
    ...);

---

// Wait for mailbox message from SPU 0 indicating data is ready.
data_addr = spu_read_in_mbox();
// DMA in data from SPU 0 local store and
// wait until complete.
mfc_get(data, data_addr, data_size, ...);
mfc_read_tag_status_all();
SPE-SPE DMA Example

// Notify SPU 0 that data has been read.
mfc_put(<garbage>,
    cb.spu_control[0] + 12,
    4,
    ...
);

// Process data
...

// Wait for acknowledgement from SPU 1.
spu_read_in_mbox();
return 0;

// Notify SPU 0 that data has been read.
mfc_put(<garbage>,
    cb.spu_control[0] + 12,
    4,
    ...
);

// Process data
...

// Wait for acknowledgement from SPU 1.
spu_read_in_mbox();
return 0;
// DMA processed data back to memory and wait
// until complete
mfc_put(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
SPE-SPE DMA Example

// Notify PPU.
spu_write_out_mbox(0);
return 0;
DMA and SPEs

- 1 Memory Flow Controller (MFC) per SPE
- High bandwidth – 16 bytes/cycle
- DMA transfers initiated using special channel instructions
- DMA transfers between virtual address space and local store
  - SPE uses PPE address translation machinery
  - Each SPE local store is mapped in virtual address space
    - Allows direct local store to local store transfers
    - Completely on chip, very fast
- Once DMA commands are issued, MFC processes them independently
  - SPU continues executing/accessing local store
  - Communication-computation concurrency/multibuffering essential for performance
DMA and SPEs

- Each MFC can service up to 24 outstanding DMA commands
  - 16 transfers initiated by SPU
  - 8 additional transfers initiated by PPU
    - PPU initiates transfers by accessing MFC through MMIO registers
- Each DMA transfer is tagged with 5-bit program-specified tag
  - Multiple DMAs can have same tag
  - SPU/PPU can wait or poll for DMA completion by tag mask
  - Can enforce ordering among DMAs with same tag
DMA Alignment

- 1/2/4/8/16-byte transfers that are naturally aligned
- Multiples of 16 bytes up to 16 KB per transfer
- DMA transfers of 16 bytes or less are atomic, no guarantee for anything else
- Memory and local store addresses must have same offset within a qword (16 bytes)
- DMA list commands
  - SPU can generate list of accesses in local store
  - Transfers between discontinuous segments in virtual address space to contiguous segment in local store
  - MFC processes list as single command
Mailboxes and Signals

- Facility for SPE to exchange small messages with PPE/other SPEs
  - e.g. memory address, “data ready” message
- From perspective of SPE
  - 1 inbound mailbox (4-entry FIFO) – send messages to this SPE
  - 1 outbound mailbox (1-entry) – send messages from this SPE
  - 1 outbound mailbox (1-entry) that interrupts PPE – send messages from this SPE to PPE
  - 2 signal notification registers – send messages to this SPE
    - Act as 1 entry or 32 independent bits
  - 32 bits
- SPU accesses its own mailboxes/signals by reading/writing to channels with special instructions
  - Read from inbound mailbox, signals
  - Write to outbound mailboxes
  - Accesses will stall if empty/full
Mailboxes and Signals

- SPE/PPE accesses another SPE mailboxes/signals through MMIO registers
  - Accesses do not stall
  - Read outbound mailboxes
  - Write inbound mailbox, signals
  - Accesses by multiple processors must be synchronized
  - If inbound mailbox overflows, last item is overwritten
  - Reading outbound mailbox when no data may return garbage
List of Useful Functions

PPU (libspe)

- spe_create_thread
- spe_wait
- spe_write_in_mbox
- spe_stat_in_mbox
- spe_read_out_mbox
- spe_stat_out_mbox
- spe_write_signal
- spe_get_ls
- spe_get_ps_area
- spe_mfc_get
- spe_mfc_put
- spe_mfc_read_tag_status
- spe_create_group
- spe_get_event

SPU

- mfc_get
- mfc_put
- mfc_stat_cmd_queue
- mfc_write_tag_mask
- mfc_read_tag_status_all/any/immediate
- spu_read_in_mbox
- spu_stat_in_mbox
- spu_write_out_mbox, spu_write_out_intr_mbox
- spu_stat_out_mbox, spu_stat_out_intr_mbox
- spu_read_signal1/2
- spu_stat_signal1/2
- spu_write_event_mask
- spu_read_event_status
- spu_stat_event_status
- spu_write_event_ack
- spu_read_decrementer
- spu_write_decrementer
Agenda

- Computing with SPEs
- SIMD
- Functional debugging
- Performance debugging
Many compute-bound applications perform the same computations on a lot of data

- Dependence between iterations is rare
- Opportunities for data parallelization

Scalar code

```java
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i]
}
```
SIMD

- Single Instruction, Multiple Data
- SIMD registers hold short vectors
- Instruction operates on all elements in SIMD register at once

Scalar code

```c
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i]
}
```

Vector code

```c
for (int i = 0; i < n; i += 4) {
    c[i] = vector_add(a[i] + b[i]);
}
```
Vector Registers

- Only registers in SPU are 128-bit registers
  - Any type (including scalar types) can go into any register
- Scalar values go in a particular position in register

- There is overhead associated with loading and storing scalars
Hardware Support for Data Parallelism

- Registers are 128-bits
- Can pack vectors of different data types into registers
- Operations consume and produce vector registers
  - Special assembly instructions
  - Access via C/C++ language extensions (intrinsics)
Writing Efficient SIMD Code

- Used the `aligned` compiler directive to control placement
  - Quadword alignment for loads and stores (`aligned(16)`)  
- Transfer multiples of 16 bytes on loads and stores
  - Pad end of data if necessary
Vector Data Types

- Vector data types dictate how to interpret 128 bits
- Available on PPU and SPU:
  - 16x 8-bit int: vector signed char
  - 8x 16-bit int: vector signed short
  - 4x 32-bit int: vector signed int
  - 4x float: vector float
- Available on SPU:
  - 2x 64-bit int: vector signed long long
  - 2x double: vector double
- Pointer types, arrays, etc. work correctly
Vector Operations

- Compilers will insert vector instructions correctly for +, *, etc. when applied to vector types
- Intrinsics provide C/C++ access to vector instructions, including many which do not correspond to any operator
  - Example: `vector signed int c = spu_add(a, b);`
  - No need to worry about registers for operands
  - Looks like a function call
  - Compiler automatically generates instructions in assembly
  - Slightly different intrinsics available on PPU, SPU
Source Headers Necessary for Intrinsics

- SPU intrinsics
  - #include <spu_intrinsics.h>
  - #include <spu_mfcio.h>
- PPU intrinsics
  - #include <ppu_intrinsics.h>
  - #include <vec_types.h>
Initializing Vectors

- One of these cast notations should work (depending on your compiler):
  - `vector signed int a = (vector signed int)(10, 20, 30, 40);`
  - `... (vector signed int){10, 20, 30, 40};`

- Or use an intrinsic:
  - `vector signed int b = spu_splats(20);`
    `// Same as (20, 20, 20, 20)`
Accessing Vector Elements

- **typedef union {**
  ```c
  int v[4];
  vector signed int vec;
  } intVec;
  ```

- **Unpack scalars from vector:**
  ```c
  intVec a;
  a.vec = ...;
  ... = a.v[2];
  ```
  ```c
  ... = spu_extract(va, 2);
  ```

- **Pack scalars into vector:**
  ```c
  a.v[0] = ...; a.v[1] = ...;
  ```
  ```c
  ... = a.vec;
  ```

Interpret a segment of memory either as an array... [v[0] v[1] v[2] v[3]]

... or as a vector type...

... so that values written in one format can be read in the other
Vector Operations

- Integer instructions
- Floating-point instructions
- Permutation/formatting instructions
- Load and store instructions
Vector Arithmetic and Logical Operations

- **PPU**
  - `vec_add, vec_sub, vec_madd, ...`
  - `vec_and, vec_or, vec_xor, ...`

- **SPU**
  - `spu_add, spu_sub, spu_madd, spu_mul, spu_re, ...`
  - `spu_and, spu_or, spu_xor, ...`

- **Rotate shifts vector elements left or right**
  - `spu_rl(v, count)`
  - `vec_rl(v, count)`

- **Integer/FP operation associated with the correct vector types (char, int, float, etc.)** is usually automatically selected by the compiler.
Vector Shuffle Operation

- **Rearrange bytes of vectors**: \texttt{spu\_shuffle(A, B, pattern)}
  - Each byte of the output is one of the bytes of \texttt{A} or \texttt{B}
  - For each byte of output, corresponding byte of pattern specifies which byte of \texttt{A} or \texttt{B} to copy
    - Bit 4 of each pattern byte specifies \texttt{A} or \texttt{B}
    - Bits 0-3 (4 low-order bits) of each pattern byte specify which byte (0-15) of source to take
    - Ex: 2\textsuperscript{nd} byte of pattern is 0x14, so take byte 4 from \texttt{B}

\[
VT = \text{spu\_shuffle}(VA, VB, VC)
\]
Vector Shuffle Operation

- Generating the shuffle pattern:

```c
pattern =
    (vector unsigned char)( b0, b1, b2, b3,
                          b4, b5, b6, b7, 
                          b8, b9, b10, b11, 
                          b12, b13, b14, b15);
```

- Example: reverse the order of bytes in `a`

```c
a = spu_shuffle(a, a,
    (vector unsigned char)(15,14,13,12,
                          11,10, 9, 8, 
                          7, 6, 5, 4, 
                          3, 2, 1, 0);
```
Exercise

- Given a vector float
  \((a, b, c, d)\)
- Return the vector float
  \((a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d)\)
- Can do this with two shuffles and two adds
Sample Implementation

vec_float4 A, B, C, D, E;
A = {...};
B = spu_shuffle(A, A,
   (vector unsigned char)(4, 5, 6, 7, 0, 1, 2, 3,
                           12, 13, 14, 15, 8, 9, 10, 11));
C = spu_add(A, B);
D = spu_shuffle(C, C,
   (vector unsigned char)(8, 9, 10, 11, 12, 13, 14, 15,
                           0, 1, 2, 3, 4, 5, 6, 7));
E = spu_add(C, D);
return E;
Agenda

- Computing with SPEs
- SIMD
- **Functional debugging**
- Performance debugging
Preparing for Debugging

- Two methods
  - Get program state on crash
  - Attach and step through program
- Compile for debugging
  - Use `gcc -g` or `xlc -g` to generate debugging info
  - or, in our Makefile:
    `CC_OPT_LEVEL = $(CC_OPT_LEVEL_DEBUG)`
Running Processes Under GDB

- **ppu-gdb ./hello-world**
  
  (gdb) run [args]

  ...

  (gdb) quit

- **export SPU_INFO=1**
  
  for extra information about threads
Attaching to Running Programs

- `ppu-gdb ./hello -p 1234`
  
  (gdb) continue

  ...

  (gdb) detach

- Finding the PID
  
  - `./hello &`
    
    [1] 1234
  
  - `ps -e | grep hello`
    
    1234 pts/2 00:00:01 hello

  - `top`
Examining Program State

- Stack trace
  - (gdb) bt

```
#0  0x0f6a7fc8 in mmap () from /lib/libc.so.6
#1  0x0f2a62e0 in pthread_create@GLIBC_2.1 () from ...
#2  0x0ff98168 in spe_create_thread () from /usr/lib...
#3  0x01801bec in calc_dist () at dist.c:36
#4  0x01801cdc in main () at dist.c:55
```
Examining Program State

- Examine variables
  - (gdb) info locals

- Evaluate expressions
  - (gdb) print VARNAME
    - (gdb) print 'FILENAME'::VARNAME
    - (gdb) print 'FUNCTION'::VARNAME
  - (gdb) print EXPR
    - Example: (gdb) print x + 100 * y

- gdb knows data types and prints values appropriately
  - To show type: (gdb) whatis VARNAME
Examining Code

- View code at a specific location
  - (gdb) list LINENUM
  - (gdb) list FUNCTION
  - (gdb) list FILENAME:FUNCTION

- Display code above/below previous snippet
  - (gdb) list
  - (gdb) list -

```
21     calc_dist()
22     {
23           speid_t id[2];
24
25     // Set up different co...
```
Controlling Program Execution

- Run to first line of main procedure
  - `(gdb) start`
- Next line in current procedure
  - `(gdb) next`
- Descend into function calls
  - `(gdb) step`
- Run until function exit, return to caller
  - `(gdb) finish`
- Resume execution until next breakpoint
  - `(gdb) continue`
- Cease debugging
  - Allow program to continue after gdb exits: `(gdb) detach`
  - Exit gdb: `(gdb) quit`
Breakpoints

- Halt program when a certain point is reached in execution
- Setting breakpoints
  - `(gdb) break FUNCTION`
  - `(gdb) break LINENUM`
  - `(gdb) break FILENAME:FUNCTION`
  - `(gdb) break FILENAME:LINENUM`
  - Conditional breakpoints:
    - `(gdb) break ... if EXPR`
    - Example expression: `(x == 1 && y == 2)`
- Viewing or removing breakpoints
  - `(gdb) info breakpoints`
  - `(gdb) remove 2`
Watchpoints

- Halt program when a value changes
- `(gdb) watch VAR`
  - `watch myVar`
  - `watch myArray[6]`
Examining Memory

- (gdb) x/Ni ADDR
- N = how many units (machine words) to show
  - Default N = 1
- Flag before address controls how to interpret data
  - i: machine instructions
  - x: hex
  - d: decimal
  - a: address (calculates offset from nearest symbol)
  - f: floating point numbers
  - s: string
Examining Memory: Example

- `const char* a = "cell-processor\n";`

- **Display as string**
  - Note that count ("1") is by strings, not words
  - `(gdb) x/ls a
  0x10000bc0 <__dso_handle+4>: "cell-processor\n"

- **Display as hex**
  - `(gdb) x/4x a
  0x10000bc0 <__dso_handle+4>: 0x63656c6c 0x2d70726f 0x657373 0x720a00
  0x63656c6c 0x2d70726f 0x657373 0x720a00
  "cell-processor\n\0"
Selecting Frames

- View state higher up in the call stack
  - Frame numbers are given by `bt`
  - `(gdb) frame 0`
    - `(gdb) frame 1`
    - `(gdb) frame 2`
    - ...
  - `(gdb) up`
    - `(gdb) down`
Debugging From emacs

- **M-x gdb** invokes gdb
  - Replace 'gdb' with 'ppu-gdb' when prompted
  - Specify executable path relative to current buffer's directory
  - Enter gdb commands in *gud-...* buffer
  - Active line in current frame is highlighted in editor

- Keyboard shortcuts available in source code files
  - Set breakpoint: `C-x SPC`
  - Print value of selected expression: `C-x C-a C-p`
  - Step: `C-x C-a C-s`  
  - Next: `C-x C-a C-n`  
  - Down frame: `C-x C-a >`  
  - Up frame: `C-x C-a <`
Exercise

- Find the value of control block (cb) in SPU thread
  - Get the recitation tarball
    - tar zxf rec4.tar.gz
  - Build the program
    - cd rec4/dma-alignment/
    - make
  - Run to the error with ppu–gdb
  - Debug
Debugging Threaded Programs

- When a new thread is entered, gdb prints
  \[\text{New Thread 123 (LWP 6041)}\]
- List threads
  - (gdb) info threads
- gdb maintains 'current thread', used for bt, etc.
  - Switch threads: (gdb) thread 2
- On breakpoint or signal, gdb makes the triggered thread current
Exercise

- Verify that \texttt{cb} in the first SPU thread is the same as \texttt{cb[0]} in the PPU program
  - You will need to qualify names
  - Build the program
    - \texttt{cd rec4/lab1/}
    - \texttt{make}
  - Set breakpoints, run and debug

```c
typedef struct {
    uintptr32_t a_addr;
    uintptr32_t b_addr;
    uintptr32_t c_addr;
    uint32_t    padding;
} CONTROL_BLOCK;
```
Exercise

(gdb) break dist_spu.c:19
(gdb) run
(gdb) print cb

$1 = {a_addr = 25286272, b_addr = 25269248, res_addr = 25269888, padding = 0}

(gdb) thread 1
(gdb) print 'dist.c'::cb

$2 = {{a_addr = 25286272, b_addr = 25269248, res_addr = 25269888, padding = 0}, {a_addr = 25286528, b_addr = 25269248, res_addr = 25278080, padding = 0}}
Exercise

- Types are consistent with source code

(gdb) whatis cb

type = CONTROL_BLOCK

(gdb) whatis 'dist.c'::cb

type = CONTROL_BLOCK [2]
Debugging Threaded Programs

- Gdb can get confused by SPU threads
  - Gdb removes breakpoint after first thread exits
  - Gdb may complain about source files for SPU program
    - "No source file named dist_spu.c. Make breakpoint pending on future shared library load? (y or [n])"
    - Choose "y" and continue, source should be visible later
Debugging SPU Threads Alone

- Use spu-gdb to debug individual SPU threads
  - `SPU_DEBUG_START=1 ./hello &`
    - Prints PIDs of threads; threads wait for debugger to attach
      "Starting SPE thread 0x181e038, to attach debugger use: spu-gdb -p 1234"
  - `spu-gdb ./spu-hello -p 1234`
    - Attach gdb to SPU thread
Troubleshooting Common gdb Issues

- **Problem: gdb examines wrong variable when names are ambiguous**
  - Use spu-gdb or rename variables
- **Problem: breakpoints are deleted prematurely**
  - Use spu-gdb or keep threads alive for as long as possible
- **Error: "Thread Event Breakpoint: gdb should not stop!"**
  - Use spu-gdb
Errors that Debugger Can Help With

- "Bus error"
  - DMA transfer problem
  - Memory misalignment
- "Segmentation fault"
  - Invalid address
- Deadlock
  - Attach and examine state
Agenda

- Computing with SPEs
- SIMD
- Functional debugging
- Performance debugging
Performance Counters on the SPUs

- Each SPU has a counter that counts down at a fixed rate (decrementer)
  - Can be used as a clock
  - Suitable for coarse-grained timing (1000s of instructions)
Decrementer Example

```c
#define DECR_MAX 0xFFFFFFFF
#define DECR_COUNT DECR_MAX

// Start counting
spu_writech(SPU_WrDec, DECR_COUNT);
spu_writech(SPU_WrEventMask, MFC_DECREMENTER_EVENT);
start = spu_readch(SPU_RdDec);
    function_of_interest();
// Stop counting, print count
end = spu_readch(SPU_RdDec);
printf("Time elapsed: %d\n", start - end);
spu_writech(SPU_WrEventMask, 0);
spu_writech(SPU_WrEventAck, MFC_DECREMENTER_EVENT);
```
Review: Instruction Scheduling

- Instructions mostly of the form $r_3 = f(r_1, r_2)$
  - Assembly file is a human-readable representation of these instructions
- Conceptually, instructions execute in the order in which they appear in assembly
Instruction Scheduling

- With pipelining, order of instructions is important!
  - Pipeline stalls while waiting for dependencies to complete

execute \( b \) before \( a \)

\[
\begin{align*}
  a & : \text{ADD} \ r3, r1, r2 \\
  b & : \text{ADD} \ r6, r4, r5 \\
  c & : \text{ADD} \ r8, r6, r7
\end{align*}
\]

- \( c \) flow dependent on \( b \)
- Assume 2 cycles operation latency
Static Profiling

- Use static profiling to see where stalls happen
- Generate assembly and instruction schedule
  - Manually
    ```
    # generate assembly (xlc -S also works)
    % gcc -S filename.c
    # generate timing information
    % /opt/ibm/cell-sdk/prototype/bin/spu_timing
      -running-count ./filename.s
    - Output stored in filename.s.timing
      - -running-count shows cycles elapsed after each instruction
    ```
  - With our Makefile
    ```
    % SPU_TIMING=1 make filename.s
    ```
Reading the Assembly

- Instructions of the form
  \texttt{OP DEST SRC1 SRC2 ...}
- Header indicates source files:

\begin{verbatim}
.file "dist_spuc.c"
.file 1 "dist_spuc.c"
.file 2 "/opt/ibmcmp/xlc/8.1/include/spu_intrinsics.h"
.LS_p1_f1_l19:
.loc 1 19 0
ila $7,a
\end{verbatim}
## Interpreting Static Profiler Output

<table>
<thead>
<tr>
<th>Pipeline No.</th>
<th>One digit for each cycle</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>129 0D 90</td>
<td>ai $6,$6,-1</td>
<td></td>
</tr>
<tr>
<td>129 1D 9012</td>
<td>cwx $12,$5,$2</td>
<td></td>
</tr>
<tr>
<td>133 1 ---3456</td>
<td>rotqby $8,$8,$10</td>
<td></td>
</tr>
<tr>
<td>134 1 4567</td>
<td>rotyqby $9,$9,$11</td>
<td></td>
</tr>
<tr>
<td>138 0D ---890123</td>
<td>fm $8,$8,$9</td>
<td></td>
</tr>
<tr>
<td>138 1D 890123</td>
<td>lqx $9,$5,$2</td>
<td></td>
</tr>
<tr>
<td>144 1 -----4567</td>
<td>shufb $8,$8,$9,$12</td>
<td></td>
</tr>
</tbody>
</table>

- **D for dual-issue**
- **-running-count** adds cycle count column
- **for stalls**

(example: rotqby requires 4 cycles to complete)
Instruction Scheduling on Cell

- In-order execution
- Dual pipeline
  - Pipeline selected based on instruction type
  - Two instructions can be issued simultaneously when dependencies allow
- Goal: scheduling instructions to minimize stalls
  - Loads, fp instructions liable to take a long time
  - Dual-issue whenever possible
  - IPC = 2 (instructions per cycles)
  - CPI = .5 (cycles per instruction)
### Example Schedule Optimization

(dist_spus.s line 246) \texttt{LS_p1_f1_l26:}

- \texttt{loc 1 26 0}
- \texttt{or $2,3,3}$
- \texttt{ila $3,\text{dist}}$
- \texttt{lqd $4,80($1)}
- \texttt{shli $4,4,8}$
- \texttt{lqd $5,96($1)}
- \texttt{shli $5,5,2}$
- \texttt{a $4,4,5$}
Example Schedule Optimization

(dist_spu.s line 246) .LS_p1_f1_l126:
   .loc 1 26 0

789012  lqd   $4,80 ($1)
890123  lqd   $5,96 ($1)
  90    or    $2,$3,$3
   01    ila   $3,dist
      --3456  shli  $4,$4,8
        4567  shli  $5,$5,2
     ---89  a     $4,$4,$5

8 cycles saved
Exercise

- Improve performance by rescheduling instructions
  - `tar zxf rec5.tar.gz`
  - `cd rec5/lab1/spu`

- Examine assembly code
  - `export CELL_TOP=/opt/ibm/cell-sdk/prototype`
  - `SPU_TIMING=1 make dist_spu.s`
  - Find an opportunity for performance gain via instruction scheduling and implement it (e.g., reduce stalls after `lqd` instructions near line 246)

- Generate object file from assembly
  - `./make-obj-file; cd ..; make`
  - `make-obj-file` compiles your modified assembly to binary, otherwise your optimization is lost

- Run and evaluate
  - How many cycles did you save?
    - `/opt/ibm/cell-sdk/bin/spu_timing -running-count dist_spu.s`
  - Is the new code correct?
    - Run and check if correctness test passes
Instruction Scheduling

- Compilers are very good at doing this automatically
  - Unoptimized code: 469 cycles
  - Optimized code (xlc -O5): 188 cycles
- Hand-reordering of optimized assembly is unlikely to produce significant gains except in extreme scenarios
Notes on Static Profiling

- Static profiler presents a skewed view of conditionals, loops
  - 8 cycles saved in the static schedule → how many cycles saved when the program runs?
- Data-dependent behavior not captured
  - Static profiler does not factor in loop trip counts or branch frequencies
  - Profiling doesn't account for branch misprediction
Improving Branch Prediction

- Static branch hinting from source code
  - `if(__builtin_expect(CONDITION, EXPECTED))`
  - Useful macros:
    - `#define LIKELY(exp) __builtin_expect(exp, TRUE)`
    - `#define UNLIKELY(exp) __builtin_expect(exp, FALSE)`
    - `if(LIKELY(i == j)) { ... }`
Performance Debugging Summary

- Static and dynamic profiling tools are used to identify performance bottlenecks

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decrementers</strong></td>
<td>Easy to set up</td>
<td>Little insight into sources of stalls</td>
</tr>
<tr>
<td>Use to measure runtime for a segment of code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Schedule analysis</strong></td>
<td>Identifies exactly where time is spent</td>
<td>Low level; only does straight-line analysis</td>
</tr>
<tr>
<td>Use to see instruction-level interactions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programming the Cell

- Guide to programming PS3/Cell: google “PS3 programming”
  - http://cag.csail.mit.edu/ps3
  - MIT short course on parallel programming using the PS3/Cell as the student project platform
  - Provides detailed examples with walk through
    - Lectures, recitations, and labs
  - Student projects and source code
  - Lots of recipes (installing Linux, SDK, Cell API mini-reference)
  - Links to additional documentation