Cell Broadband Engine
Powering PS3s and Blades

Rodric Rabbah
IBM Watson Center
rabbah@mit.edu
Performance at all costs

- Not just people in national labs… but game developers also
  - Yes it’s hard, so what? You have to do it!

- Experience with Cell has demonstrated that good programming models are not optional for multicores
Parallelism Applicable Everywhere: can be fun!
Cell Broadband Engine

- Versatile high-performance computing platform
  - Graphics, Scientific, Parsing (XML, Web browsers), Black Scholes option pricing, Network Processing, H.264 MPEG4 encoding/decoding, Bioinformatics

- Heterogeneous processor with 9 cores
- 1 general purpose core: Power Processor Element (PPE)
- 8 accelerator cores: Synergistic Processor Elements (SPEs)
Cell Broadband Engine Architecture

64-bit Power Architecture w/ VMX

L1

L2

SPU

PPU

MIC

BIC

16B/cycle

32B/cycle

16B/cycle (2x)

16B/cycle
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)
- 32KB L1 cache (I & D) and 512KB L2
Synergistic Processor Element

- Specialized high performance core
- Three main components
  - SPU: processor
  - LS: local store memory
  - MFC: memory flow control manages data in and out of SPE
SPU Processing Core

- In-order processor: no speculation or branch prediction

- Greatest compute power is single precision floating point
  - Single precision floating point is not full IEEE compliant, similar to graphics HW
  - Double precision floating point is full IEEE compliant

- 128 unified registers used for all data types

- Can only access (load & store) data in the SPE local store
SPEs vs GPU

- ~4-5x better performance from SPEs compared to GPUs
- Renewed interest in ray tracing: finally practical for real time
- Visualization of huge digital models
- 720p and 1080p HDTV Output
- Seamless Scale Out
  - more cores $\rightarrow$ more performance
- Real-time ambient occlusion
- Dynamic load balancing

Courtesy of Barry Minor, IBM
Compute Hierarchy

SPE Screen Partitions

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

QS20 Blade, Two Cell Processors, 16 SPEs
Dynamic Compute Hierarchy

SPE Partitions

Blade 0
Blade 1
Blade 2
Blade 3
Blade 4
Blade 5
Blade 6

Seven QS20 Blades, 3.2 TFLOPS
Local Store (LS)

- 256KB of memory per SPE
- Code and data share LS
- SPU can load 16B per cycle from LS

- Data from main memory is explicitly copied to and from the local store since SPU cannot access any other memory locations directly
Data In and Out of the SPE Local Store

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

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

- SPU needs data
  1. SPU initiates MFC request for data
  2. MFC requests data from memory
  3. Data is **copied** to local store
Data In and Out of the SPE Local Store

- SPU needs data
  1. SPU initiates MFC request for data
  2. MFC 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 Local Store

- SPU needs data
  1. SPU initiates MFC request for data
  2. MFC 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
MFC DMAs and SPEs

- 1 Memory Flow Controller (MFC) per SPE
- High bandwidth – 16B/cycle
- 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
- 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
Element Interconnect Bus

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

The good and the hard
What Makes Cell Diff*?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation
- SPEs and SPE local store
  - Something new to worry about, but potential for more efficiency
- Short vector parallelism (SIMD)
  - Intra-core parallelism in addition to parallelism across cores
SPU Programs

- SPU programs are designed and written to work together but are compiled independently
- Separate compiler and toolchain (ppu-gcc and spu-gcc)
- Produces small ELF image for each program that can be embedded in PPU program
  - Contains own data, code sections
  - On startup, C runtime (CRT) initializes and provides malloc
  - printf/mmap/some other I/O functions are implemented by calling on the PPU to service the request
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;
}
```
Mapping Computation to SPEs

- Original single-threaded program performs many computation stages on data

- How to map to SPEs?
Mapping Computation to SPEs

- Coarse-Grained Data Parallelism
  - Each SPE contains all computation stages
  - Split up data and send to different SPEs
Example Data Parallelization on Cell

- Calculate distance from each point in a[...] to each point in b[...] and store result in c[...][...]

for (i = 0; i < NUM_POINTS; i++) {
    for (j = 0; j < NUM_POINTS; j++) {
        c[i][j] = distance(a[i], b[j]);
    }
}

- How to divide the work between 2 SPEs?
Example Data Parallelization on Cell

Same code for each SPU
Mapping Computation to SPEs

- **Coarse-Grained Pipeline Parallelism**
  - Map computation stages to different SPEs
  - Use DMA to transfer intermediate results from SPE to SPE in pipeline fashion
Mapping Computation to SPEs

- Mixed approaches, other approaches are possible
- Depends on problem

![Diagram showing mapping of computation to SPEs]
Increasing Performance with Parallelism

What’s all the fuss about?
Cell-ifying a Program

- Simple 3D gravitational body simulator
- $n$ objects, each with mass, initial position, initial velocity

```c
float mass[NUM_BODIES];
VEC3D pos[NUM_BODIES];
VEC3D vel[NUM_BODIES];
```

- Simulate motion using Euler integration
Single-threaded Version

- For each step in simulation
  - Calculate acceleration of all objects
    - For each pair of objects, calculate the force between them and update accelerations accordingly
  - Update positions and velocities

- Slow: $n = 3072 \rightarrow 1500\text{ms}$

VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES - 1; i++) {
    for (j = i + 1; j < NUM_BODIES; j++) {
        // Displacement vector
        VEC3D d = pos[j] - pos[i];
        // Force
        t = 1 / sqr(length(d));
        // Components of force along displacement
        d = t * (d / length(d));

        acc[i] += d * mass[j];
        acc[j] += -d * mass[i];
    }
}

VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES; i++) {
    pos[i] += vel[i] * TIMESTEP;
    vel[i] += acc[i] * TIMESTEP;
}
Cell-ification: using SPEs for acceleration

- Divide objects into 6 sections \( n = 3072 = 6 \times 512 \)
- Each SPU is responsible for calculating the motion of one section of objects
  - SPU still needs to know mass, position of all objects to compute accelerations
  - SPU only needs to know and update velocity of the objects it is responsible for
- Everything fits in local store
  - Positions for 3072 objects take up 36 KB
Cell-ification: using SPEs for acceleration

- **Initialization**
  - PPU tells SPU which section of objects it is responsible for

```c
// Pass id in envp
id = envp;
own_mass = mass[id];
own_pos = pos[id];

// Index [i] stores mass/position of objects SPU i
// is responsible for
float mass[6][SPU_BODIES];
VEC3D pos[6][SPU_BODIES];

// The section of objects this SPU is responsible for
int id;
// Pointer to pos[id]
VEC3D *own_pos;
// Velocity for this SPU's objects
VEC3D own_vel[SPU_BODIES];
```
Cell-ification: using SPEs for acceleration

- SPU copies in mass of all objects

```c
mfc_get(mass, cb.mass_addr, sizeof(mass), ...);
```

- SPU copies in initial position, velocity of its objects

```c
mfc_get(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
mfc_get(own_vel, cb.vel_addr + id * sizeof(own_vel), sizeof(own_vel), ...);
```
Cell-ification: using SPEs for acceleration

- Simulation step
  - PPU sends message telling SPU to simulate one step

```c
spu_read_in_mbox();
```

Diagram:
- PPU/Memory
  - `pos`
- SPU 2
  - `pos`
  - `vel`
Cell-ification: using SPEs for acceleration

- SPU copies in updated positions of other objects

```c
if (id != 0) {
    mfc_get(pos, cb.pos_addr + id * sizeof(pos[0]), id * sizeof(pos[0]), ...);
}
if (id != 5) {
    mfc_get(pos[id + 1], cb.pos_addr + (id + 1) * sizeof(pos[0]),
            (5 - id) * sizeof(pos[0]), ...);
}
```
Cell-ification: using SPEs for acceleration

- SPU sends message to PPU indicating it has finished copying positions
  - PPU waits for this message before it can tell other SPUs to write back positions at end of simulation step

```c
spu_write_out_mbox(0);
```
Cell-ification: using SPEs for acceleration

- SPU calculates acceleration and updates position and velocity of its objects

```c
// Process interactions between this SPU's objects
process_own();
// Process interactions with other objects
for (int i = 0; i < 6; i++) {
  if (i != id) {
    process_other(pos[i], mass[i]);
  }
}
```

- Diagram showing PPU/Memory and SPU 2 with position (pos) and velocity (vel) states.
SPU waits for message from PPU indicating it can write back updated positions

```c
spu_read_in_mbox();
```
Cell-ification: using SPEs for acceleration

- SPU writes back updated positions to PPU

```
mfc_put(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
```
SPU sends message to PPU indicating it is done simulation step

```c
spu_write_out_mbox(0);
```
Coordination with 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) – interrupts PPE to send messages from SPE
  - 2 signal notification registers – send messages to this SPE
  - 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

- SPE/PPE accesses another SPE mailboxes/signals through MMIO registers
Orchestration and Coordination

- Lots of signals sent back and force
  - I’m ready
  - I’m done
  - What’s my work?
  - Where’s my data?
  - …

- Couple this with architecture issues
  - Cell alignment constraints

- And a lot can go wrong
  - Many existing methodologies for debugging do not scale well for multicores
Cell-ified Version

- Acceleration calculation
  - In single-threaded version, we calculate distance between each pair of objects once
  - In Cell-ified version, if 2 SPUs are responsible for objects $i$ and $j$, each SPU will calculate the distance separately

- With 6 SPEs, what speedup can we expect?
Overlapping Communication and Computation
Overlapping DMA and Computation

- Simple approach:
  - Pipelining can achieve communication-computation concurrency
    - Start DMA for next piece of data while processing current piece

Synchronization point
// pos[i] stores positions of objects SPU i is responsible for
VECD pos[6][SPU_BODIES];

// Start transfer for first section of positions
i = 0;
tag = 0;

mfc_get(pos[i],
    cb.pos_addr + i * sizeof(pos[0]),
    sizeof(pos[0]),
    tag,
    ...);
tag ^= 1;

// Process interactions between objects this SPU is responsible for
process_own();
while (!done) {
    // Start transfer for next section of positions
    mfc_get(pos[next_i],
            cb.pos_addr + next_i * sizeof(pos[0]),
            sizeof(pos[0]),
            tag,
            ...);

    // Wait for current section of positions to finish transferring
    tag ^= 1;
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();

    // Process interactions
    process_other(pos[i], mass[i]);
    i = next_i;
}
while (!done) {
    // Start transfer for next section of positions
    mfc_get(pos[next_i],
            cb.pos_addr + next_i * sizeof(pos[0]),
            sizeof(pos[0]),
            tag,
            ...);

    // Wait for current section of positions to finish transferring
    tag ^= 1;
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();

    // Process interactions
    process_other(pos[i], mass[i]);

    i = next_i;
}

while (!done) {
    // Start transfer for next section of positions
    mfc_get(pos[next_i],
        cb.pos_addr + next_i * sizeof(pos[0]),
        sizeof(pos[0]),
        tag,
        ...);

    // Wait for current section of positions to
    // finish transferring
    tag ^= 1;
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();

    // Process interactions
    process_other(pos[i], mass[i]);
    i = next_i;
}
while (!done) {
    // Start transfer for next section of positions
    mfc_get(pos[next_i],
            cb.pos_addr + next_i * sizeof(pos[0]),
            sizeof(pos[0]),
            tag,
            ...);

    // Wait for current section of positions to
    // finish transferring
    tag ^= 1;
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();

    // Process interactions
    process_other(pos[i], mass[i]);

    i = next_i;
}
// Wait for last section of positions to finish
// transferring
tag ^= 1;
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_all();

// Notify PPU that positions have been read
spu_write_out_mbox(0);

// Process interactions
process_other(pos[i], mass[i]);
Overlapping DMA and Computation

- Pipelining can improve performance by a lot, or not by much
  - Depends on program: communication to computation ratio
  - Can avoid optimizing parts that don’t greatly affect performance
Double-buffering

- LS is finite
- Avoid wasting local store space
- Keep 2 buffers
  - Start data transfer into one
  - Process data in other
  - Swap buffers for next transfer
Double-buffering
Double-buffering
Double-buffering
Intra-Core Parallelism

SIMD Programming on Cell
Many compute-bound applications perform the same computations on a lot of data

- Dependence between iterations is rare
- Opportunities for data parallelization

Scalar code

```c
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:i+3] = a[i:i+3] + b[i:i+3]
}
```
SIMD

- Can offer high performance
  - Single-precision multiply-add instruction: 8 flops per cycle per SPE
- Scalar code works fine but only uses 1 element in vector
- SPU loads/stores on quad-word (qword) granularity only
  - Can be an issue if the SPU and other processors (via DMA) try to update different variables in the same qword
- For scalar code, compiler generates additional instructions to rotate scalar elements to the same slot and update a single element in a qword
- SIMDizing code is important
  - Auto SIMDization (compiler optimization)
  - Intrinsics (manual optimization)
Example: Scalar Operation

\[ C[0] = A[0] * B[0] \]
Example: SIMD Vector Operation

\[ \text{for}(i = 0; i < N/4; ++i) \]
\[ C[i] = \text{vector\_mul}(A[i], B[i]); \]
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)
Accessing Vector Elements

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

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

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

Interpret a segment of memory either as an array…

```
```

… or as a vector type…

`vec`

so that values written in one format can be read in the other.
Review: 3D Gravitational Simulator

- \( n \) objects, each with mass, initial position, initial velocity

```c
float mass[NUM_BODIES];
VEC3D pos[NUM_BODIES];
VEC3D vel[NUM_BODIES];
```

- Simulate motion using Euler integration
  - Calculate the force of each object on every other
  - Calculate net force on and acceleration of each object
  - Update position

```c
typedef struct _VEC3D {
  float x, y, z;
} VEC3D;

VEC3D d;
// Calculate displacement from i to j
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```
Re-engineering for SIMD

- One approach to SIMD: array of structs
  - Pad each (x, y, z) vector to fill a qword
  - Components (x, y, z) correspond to first three words of vector float
  - Qwords for different vectors stored consecutively

```
typedef union _VEC3D {
    struct {float x, y, z;};
    vector float vec;
} QWORD_ALIGNED VEC3D;
```
Re-engineering for SIMD

- Now we can replace component-wise addition, subtraction, and multiplication with SIMD instructions

```c
VEC3D d;
// Calculate displacement from i to j
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```

```c
vector float d;
// Calculate displacement from i to j
d = spu_sub(pos[j].vec, pos[i].vec);
```
SIMD Design Considerations

- Data layout: struct of arrays (SOA) vs. array of structs (AOS)
  - SOA layout is alternative data organization
    - Lay out fields consecutively
  - Can apply different algorithms on new data layout

Array of structs

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>y0</td>
<td>z0</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>y1</td>
<td>z1</td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td>y2</td>
<td>z2</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td>y3</td>
<td>z3</td>
<td></td>
</tr>
<tr>
<td>x4</td>
<td>y4</td>
<td>z4</td>
<td></td>
</tr>
<tr>
<td>x5</td>
<td>y5</td>
<td>z5</td>
<td></td>
</tr>
</tbody>
</table>

Struct of arrays

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
</tr>
<tr>
<td>x4</td>
<td>x5</td>
<td>x6</td>
<td>x7</td>
</tr>
<tr>
<td>y0</td>
<td>y1</td>
<td>y2</td>
<td>y3</td>
</tr>
<tr>
<td>y4</td>
<td>y5</td>
<td>y6</td>
<td>y7</td>
</tr>
<tr>
<td>z0</td>
<td>z1</td>
<td>z2</td>
<td>z3</td>
</tr>
<tr>
<td>z4</td>
<td>z5</td>
<td>z6</td>
<td>z7</td>
</tr>
</tbody>
</table>
Struct of Array Layout

- Need 12 qwords to store state for 8 objects
  - x, y, z position and velocity components
  - No padding component needed in SOA
- For each component, do four pair-interactions at once with SIMD instructions
  - Rotate qword 3 more times to get all 16 pair-interactions between two qwords
Performance Summary for Example

- Baseline native code was sequential and scalar
  - Scalar (PPU): 1510 ms
- Parallelized code with double buffering for SPUs
  - Scalar (6 SPUs): 420 ms
- Applied SIMD optimizations
  - SIMD array of structs: 300 ms
- Redesigned algorithm to better suite SIMD parallelism
  - SIMD struct of arrays: 80 ms

- Overall speedup compared to native sequential execution
  - Expected: ~ 24x (6 SPUs * 4 way SIMD)
  - Achieved: 18x*  

* Note comparison is PPU to 6 SPU
Summary

- Programming multicore architectures: “parallelize or perish”
- Orchestrating parallelism is hard
  - Data management
  - Code placement
  - Scheduling
  - Hiding communication latency
- Lots of opportunities for impact
  - Scheduling ideas
  - Dynamic load balancing
  - Static scheduling
  - Intra-core performance still matters
- Cell offers a unique platform to explore and evaluate lots of ideas
  - PS3s make it easily accessible
Programming the Cell

- Provides detailed examples with walk through
- Lots of recipes
- Links to additional documentation