APPLICATION IMPLEMENTATION ON THE CELL B.E PROCESSOR: TECHNIQUES EMPLOYED

John Freeman, Diane Brassaw, Rich Besler, Brian Few, Shelby Davis, Ben Buley

Black River Systems Company Inc.
162 Genesee St. Utica, NY 13501
IBM Cell BE Processor

- **Cell BE processor boasts nine processors on a single die**
  - 1 Power® processor
  - 8 vector processors
- **Computational Performance**
  - 205 GFLOPS @ 3.2 GHz
  - 410 GOPS @ 3.2 GHZ
- **A high-speed data ring connects everything**
  - 205 GB/s maximum sustained bandwidth
- **High performance chip interfaces**
  - 25.6 GB/s XDR main memory bandwidth

**Excellent Single Precision Floating Point Performance**
Synergistic Processor Element (SPE)

256K Local Store

128 x 128-bit Registers

Dual-Issue Core (with special purpose pipelines)

Even

Odd

Memory Flow Controller (MFC)

SIMD Unit

++/--

**
Cell Hardware Options
Sony’s PlayStation 3

- 256MB, 40Gb, 3.2GHz, 6 SPEs
- Low cost development station $399
- Choice of Linux distribution
  - Fedora
  - Ubuntu
  - Yellowdog
- Complete binary compatibility with other Cell Platforms

Great Software Development Resource
Mercury Cell Accelerator Board 2

- 1 Cell Processor 2.8GHz
- 4GB DDR2, GbE, PCI-Express 16x, 256MB DDR2
- Full Mercury MultiCore Plus SDK Support

Workstation Accelerator Board
Mercury’s 1U Server DCBS-2

- Dual Cell Processor, 3.2GHz, 1GB XDR per Cell, Dual Gige, Inifiband/10GE Option, 8 SPE per Cell.
- Larger Memory Footprint Compared to the PS-3
- Dual Cell processor give a single application access to 16 SPEs
- Preconfigured with Yellowdog Linux
- Binary compatible with PS-3

Excellent Small Application Option
IBM QS-22 Blade

- Dual IBM PowerXCell™ 8i (New Double Precision SPEs)
- Up to 32GB DDRII Per Cell
- Dual GigE and Optional 10GE/Inifiband
- Red Hat Enterprise Linux 5.2
- Full Software/Hardware support form IBM
- Up to 14 Blades in Blade Center Chassis
- Very high density solution

Double Precision Workhorse
SONY ZEGO BCU-100

- 1U Cell Server
- Single 3.2GHz Cell/B.E. 1GB XDR, 1GB DDR2, RSX GPU, GE
- Full Cell/B.E. with 8 SPEs
- PCI-Express slot for Inifiband or 10GbE
- Pre loaded with Yellow Dog Linux

New Product
Software Development
Initial Cell Development System

- Sony Playstation 3
  - 1 3.2GHz Cell Chip
    - 6 available SPEs
  - 256MB XDR RAM
  - 20GB Hard Drive (10GB usable)
  - Gigabit Ethernet and USB connectivity
  - Yellow Dog Linux 5.0 installed
- IBM SDK 2.0 installed
- GNU GCC and associated debuggers
- Rebuilt Linux kernel
  - Allow additional networking options (to ease development)
  - Trim unneeded sections of Linux kernel
  - Different memory performance options

- 2 Week initial software/hardware setup (to achieve current configuration)
- $499 base hardware cost, $70 Linux (for early access)
Pick Your Linux

- Several Linux Distribution Available for the Cell
  - Fedora
    - IBM tools appear on Fedora first
    - Excellent Community Support
  - Yellowdog
    - Good cluster management tool
    - Default distribution for Sony and Mercury
  - Red Hat Enterprise Linux
    - Stable Distribution with Long Term Support
    - Can purchase full IBM SDK for support on Red Hat

We Have Had Excellent Results with Fedora
Software Development

- **Utilize Free Software**
  - IBM Cell Software Development Kit (SDK)
    - C/C++ libraries providing programming models, data movement operations, SPE local store management, process communication, and SIMD math functions
  - Open Source Compilers/Debuggers
    - GNU and IBM XL C/C++ Compilers
    - Eclipse IDE enhancements specifically for Cell targets
    - Instruction level debugging on PPE and SPEs
  - IBM System Simulator: Allows testing Cell applications without Cell hardware
  - Code optimization tools
    - Feedback Directed Program Restructuring (FDPR-Pro) optimizes performance and memory footprint
  - Eclipse Integrated Development Environment (IDE)
    - Compilation from Linux workstations, run remotely on Cell targets
    - Develop, compile, and run directly on Cell based hardware and System simulators
  - Linux Operating System
    - Customizable kernel
    - Large software base for development and management tools

- **Additional Software Available for purchase**
  - Mercury’s MultiCore Framework, PAS, SAL

Multiple Software Development Options Allow Greater Flexibility and Cost Savings
Application Development Using Mercury MCF

Advantages
- Node management is easy to setup and change dynamically.
- Simplifies complex data movement
- Various data I/O operations are hidden from the user after initial setup.
  - Multiple striding, overlapping, and multi-buffer options available.
- Technical Support Provided

Disadvantages
- Plugins must be explicitly defined and loaded at runtime
- SPE affinity are not supported.
- Added overhead
- Mailbox communication is restricted between a PPE and SPEs
- Single one-time transfers can be performed via IBM DMA APIs
- SPE to SPE transfers isn't supported in MCF 1.1.
- Added Overhead
- *Cost
- Possible interference when trying to utilize IBM SDK features that aren't exposed via MCF APIs.

Data processing and task synchronization are comparable between the Mercury MCF and IBM SDK
Application Development Using IBM SDK

**Advantages**
- SPE Affinity supported (SDK 2.1+)
- Plugins are implicitly loaded at run-time
- Light weight infrastructure
- SPE-SPE Communications possible
- Low Level DMA operations can be functionally hidden
- SPE-SPE Transfers are possible
- Free w/o Support

**Disadvantages**
- Lower level SPE/PPE setup/control
- Low Level DMA control and monitoring increases complexity
- Manual I/O buffer management
- Technical Support unknown

Data processing and task synchronization are comparable between the Mercury MCF and IBM SDK.
Loop Unrolling
Benefits/Effects of Loop Unrolling

- SPEs are not branch friendly
- Large register count on SPEs makes loop unrolling attractive
- Replace data independent inner loop calculations with compile-time equivalents
  - Loop index increments
  - Array index calculations based on state of shift registers
  - Lots of bit-wise operations (shift, mask, and then sum) that are data independent but are loop iteration dependent
- Creates more code to fetch
  - Larger SPE image meaning less storage for data and more local store memory access into code space
Loop Unrolling

- Inner Loop
  - Clock shift register
  - Mask bits A
  - Mask bits B
  - Sum masked bits A
  - Load data[B]
  - Load data[A]
  - Calculate (data[A], data[B])
  - Increment counter

- Compile-time (Inner Loop)
  - Clock shift register
  - Mask bits A
  - Mask bits B
  - Sum masked bits A
  - Increment counter

- Run-time (No Inner loop)
  - Load data[Aₙ]
  - Load data[Bₙ]
  - Calculate (data[Aₙ], data[Bₙ])

- Unroll Loop

Iteration-dependent operation
Data-dependent operation
Loop Unrolling for Cell Processor (SPE)

Using C++ Template Metaprogramming

Recursive Functions with Templates

Generic Template Function

Template < int I >

Vector Code

Recursive Call

(I-1)

Explicit Specialization

Template < 0 >

Vector Code

Compiled Code

Create instance with I=3

Vector Code A (I=3)

Vector Code A (I=2)

Vector Code A (I=1)

Vector Code A (I=0)
Loop Unrolling for Cell Processor (SPE)

Using C++ Template Metaprogramming

General Template Class

```
template< int STATE >
class machine
{
private:
enum { NEXT_STATE = ( STATE - 1 ) };
enum { INDEX = ((STATE << 4) + (STATE & 0xF)) };
public:
    static inline void process( float * data )
    {
        spu_vector_code( data, INDEX );
        machine< NEXT_STATE >::process( data );
    }
};
```

Explicit Specialization Template Class (recursion termination)

```
template<>
class machine<0>
{
private:
    enum { INDEX = ((STATE << 4) + (STATE & 0xF)) };
public:
    static inline void process( float * data )
    {
        spu_vector_code( data, INDEX );
    }
};
```

Usage of template classes

```
int main(int argc, char * argv[])
{
    float data[SOME_SIZE];
    machine<7>::process(data);
}

int main(int argc, char * argv[])
{
    float data[SOME_SIZE];
    machine<7>::process(data);
}
```

Expands to...

```
    spu_vector_code( data, ((7 << 4) + (7 & 0xF)) );
    spu_vector_code( data, ((6 << 4) + (6 & 0xF)) );
    spu_vector_code( data, ((5 << 4) + (5 & 0xF)) );
    spu_vector_code( data, ((4 << 4) + (4 & 0xF)) );
    spu_vector_code( data, ((3 << 4) + (3 & 0xF)) );
    spu_vector_code( data, ((2 << 4) + (2 & 0xF)) );
    spu_vector_code( data, ((1 << 4) + (1 & 0xF)) );
    spu_vector_code( data, ((0 << 4) + (0 & 0xF)) );
```
Explicit Algorithm Partitioning

```java
public static void part_a(ASMOutput out, String suffix) throws IOException {
    out.LQX("input_data"+suffix, "input", "ix");
    out.AI("ix", "ix", 16);
}

public static void part_b(ASMOutput out, String suffix) throws IOException {
    out.A("output_data"+suffix, "input_data"+suffix, "input_data"+suffix);
}

public static void part_c(ASMOutput out, String suffix) throws IOException {
    out.STQX("output_data"+suffix, "output", "ox");
    out.AI("ox", "ox", 16);
    out.AI("nPts", "nPts", -16);
}
```

Expands to...

```
LQX{ input_data_0, input, ix }
AI( ix, ix, 16 )
    LQX{ input_data_1, input, ix }
    AI( ix, ix, 16 )
    A( output_data_0, input_data_0, input_data_0 )
    HBRR( loop_br_0, loop )
    LOOP_LABEL( loop )
    LQX{ input_data_2, input, ix }
    AI( ix, ix, 16 )
    A( output_data_1, input_data_1, input_data_1 )
    STQX{ output_data_0, output, ox }
    AI( ox, ox, 16 )
    A( nPts, nPts, -16 )
    BRZ( npts, loop_br_0 )
    LQX{ input_data_0, input, ix }
    AI( ix, ix, 16 )
    A( output_data_2, input_data_2, input_data_2 )
    STQX{ output_data_1, output, ox }
    AI( ox, ox, 16 )
    A( nPts, nPts, -16 )
    BRZ( npts, loop_br_0 )
    LQX{ input_data_1, input, ix }
    AI( ix, ix, 16 )
    A( output_data_0, input_data_0, input_data_0 )
    STQX{ output_data_2, output, ox }
    AI( ox, ox, 16 )
    A( nPts, nPts, -16 )
    LABEL( loop_br_0)
    BRNZ( npts, loop )
```
Parallelization
Parallelization Techniques

- Make use of POSIX Threads to Manage SPE Resources.
- Experimented With Various Techniques
  - Round Robin
  - Fixed Function / Pipelining
  - Load Balancing
- Generally Good Results With all Techniques
  - Typically Use a Mixed Approach using Groups of SPEs
Each processor performs same tasks but on a different part of the data set.
Each processor has a dedicated task. In this design a complicated algorithm can be broken down into basic functions that are distributed to different SPEs.
Each processor can perform different tasks. When a processor becomes available it changes functionality to fit the current need of the next data block.
SPE Local Store Management

1. Move first data set to the A buffer in Local Store from main memory.

2. Move second data set to the B buffer in the Local Store from main memory while the SPU processes the data in buffer A and storing the results in output buffer 1.

3. Transfer the third input data set to buffer A where the first data set used to be. Meanwhile transfer the output data buffer 1 back to main memory.

4. Move second data set to the B buffer on the Local Store from main memory while the SPU processes the data in buffer A and storing the results in output buffer 1. Transfer the second result set back from buffer 2.
Large Programs on SPEs

*Using overlays to overcome the 256KB Local Store limitation*

- General Purpose Processor code
- Code broken into parts based on data locality
  - Function A
  - Function B
  - Function C
  - Function D
  - Function E
  - Function F
- Code and data segments are combined into single overlay
  - Functions A, B, C
  - Functions D, E
- Overlays are swapped in/out from main memory as needed

- Data
- Functions A, B, C
- Functions D, E
- Function F

- SPE
- Code
- 256 KB Local Store
Performance Metrics
<table>
<thead>
<tr>
<th>No.</th>
<th>Event Pipeline</th>
<th>Order Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>23</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>24</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>26</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>27</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>28</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>29</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>30</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
<tr>
<td>31</td>
<td>48 mfp4, 46, 36, 33</td>
<td>x</td>
</tr>
</tbody>
</table>

IBM’s ASMVis

- Use the output of IBM’s spu_timing tools
- Visualize both instructions pipelines on a SPE
- Very useful for identifying stalls in SPE code
# include "tutil.h"

/* initialize timing */
tu_init();  TU_BEG(TU_ALL);   /* times entire program */
/* Initialize logic here */

...TU_END(TU_INIT);
TU_BEG(TU_FCN1);   /* times just the initialize portion */
/* Function 1 logic here */

...TU_BEG(TU_RD);
/* File read logic here */

...TU_END(TU_RD);
TU_END(TU_FCN1);
TU_END(ALL);

/* print timing */
tu_print();

Clock resolution = 0.000000013 sec
avg clk overhead = 0.000000439 sec
thread 0x7f7fec000

all => 1 pass in 10.474993 sec = 10.474993170 sec/pass
init => 1 pass in 0.474993 sec = 0.474992999 sec/pass
function 1 => 100 passes in 10.000000 sec = 0.10000000 sec/pass
read => 100 passes in 4.000000 sec = 0.04000000 sec/pass

Library to Instrument PPE and SPE Code for High Resolution Profiling