And now for something completely different...
CorePy: High-Productivity Cell/B.E. Programming

Chris Mueller, Ben Martin, and Andrew Lumsdaine
Indiana University
June 19, 2007
Georgia Tech, Sony/Toshiba/IBM Workshop on Software and Applications for the Cell/B.E. processor
Cell/B.E.
For all your multi-core and SIMD programming needs…

⇒ Advanced “make” build system!
⇒ Cutting edge “gdb” debugger!
⇒ Unparalleled C standard library!
⇒ Works with any text editor!

* Auto-parallelizing, auto-simdizing, optimizing compiler not yet available.
  For maximum SIMD performance, use of assembly may be required.
  Void where prohibited, prohibited where void.
For all your multi-core and SIMD programming needs…

C

With Intrinsics!

⇒ Featuring *

⇒ Advanced “make” build system!
⇒ Cutting edge “gdb” debugger!
⇒ Unparalleled C standard library!
⇒ Works with any text editor!

Is there an alternative?

*Auto-parallelizing, auto-simdizing, optimizing compiler not yet available.
For maximum SIMD performance, use of assembly may be required.
Void where prohibited, prohibited where void.
The Approach

Take a modern programming technique…

(Python + C + agile development)
The Approach

Take a modern programming technique…

…provide direct access to the hardware…
The Approach

Take a modern programming technique…

…provide direct access to the hardware…

…and let programmers explore the SIMD and multi-core design spaces.

(power to the people?)
The Approach, Illustrated

[Diagram showing the process from Python to Cell/B.E.]
Huh?!? Python for Cell Programming?

- Scripting languages increase developer productivity
  - Dynamic type systems
  - Large standard library collections (~240 w/Python)
  - Concise syntax
  - “Edit and execute”
  - Limit reliance on external tools

- Scripting languages are already used in HPC
  - “Glue” code to tie applications together
  - “Kleenex” applications
  - Native libraries for performance
CorePy

CorePy is a library for creating and executing PowerPC, VMX, and SPU programs from Python.

- Execute arbitrary SPE programs from Python
- Talk to SPE programs using libspe wrappers
- Create new SPE or PPE programs directly from Python
CorePy Components

Execution Management

- InstructionStream
  Container for SPE/PPE programs
- Processor
  Execution management and libspe functions
- Memory
  Memory alignment and transfer utilites

Code Synthesis

- Instruction Set Architecture (ISA)
  PowerPC, VMX, and SPU instruction interface
- Variables
  Library-defined “primitive” types
- Iterators
  Loop generation and optimization
**Example: Population Count**

Population count (popc) counts the number of ‘1’ bits in a bit vector. It is used in many data clustering and machine-learning algorithms to compare feature vectors.

```
1. v = {0b0101010111101101, 0b1100110000000001,
      0b0000001000100001, 0b1111111111000001}
2. c = popc(v)
3. print c
4. -> 29
```

**Structure**

![Example molecule structure]

**Bit-vector Representation**

<table>
<thead>
<tr>
<th>Alkoloid bit</th>
<th>Rule-of-five bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>001011111011001</td>
<td>0000100111111...</td>
</tr>
</tbody>
</table>

**Metrics**

\[
D_{Rand} = \frac{c + d}{a + b + c + d}
\]

\[
D_{Jaccard} = \frac{c}{a + b + c}
\]

x: 10111101  
\[ x_{Rand} = .444, D_{Jaccard} = .375 \]

y: 110011010  
\[ y_{Rand} = .444, D_{Jaccard} = .375 \]
An SPE Population Count

128-bit bit vector population count using SPE instructions.

- **cntb**: count bits in bytes
- **sumb**: sum bytes into halfwords
- **a**: add
- **rotqbyi**: rotate quadword by bytes, immediate

### Example

<table>
<thead>
<tr>
<th>cntb t, v</th>
<th>2 2 3 3 (x4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sumb t, t, zero</td>
<td>(0) 10 (x4)</td>
</tr>
<tr>
<td>a c, c, t</td>
<td>10 5 3 11</td>
</tr>
<tr>
<td>rotqbyi t, t, 4</td>
<td>5 3 11 10 (reduce 3 more times)</td>
</tr>
</tbody>
</table>

Count the bits and add/reduce the bytes in each word

Add/reduce the results into the final count using add and rotate.
**spu_popc: Population Count in C**

1. #include "corepy.h"

2. int main(unsigned long long id) {
3.   // Load the vector from input parameter 3
4.   vector unsigned int x = get_vector_param_3();
5.   vector unsigned int count = (vector unsigned int){0,0,0,0};
6.   vector unsigned int result = (vector unsigned int){0,0,0,0};
7.   int i = 0;

8.   // Initialize CorePy run time
9.   spu_ready();

10.  count = (vector unsigned int)spu_cntb((vector unsigned char)x);
11.  count = (vector unsigned int)spu_sumb((vector unsigned char)count,
12.    (vector unsigned char)0);

13.   for(i; i < 4; i++) {
14.     result = result + x;
15.     x = si_rotqbyi(x, 4);
16.   }
17.  // Report the result using the out mailbox
18.  spu_write_out_mbox(spu_extract(result, 0));
19.  return SPU_SUCCESS;
20. }

The Processor object executes SPE programs and retrieves return values.

1. import corepy.arch.spu.platform as env

2. # Load the popc SPE program
code = env.NativeInstructionStream("spu_popc")
proc = env.Processor()

5. # Set the input parameters
params = env.spu_exec.ExecParams()
params.v3 = [0xDEAD, 0xBEEF, 0x1337, 0xCAFE]

8. # Execute the popc program with the parameters
count = proc.execute(code, 'mbox', params)
Processor and InstructionStream

**InstructionStream** manages executable instruction sequences

```
# Use NativeInstructionStream for compiled SPE programs
env.NativeInstructionStream("spu_popc")
```

**Processor** manages execution and return values.

Return values:
1. # result in gp_return
2. r = p.execute(c)
3. # result in fp_return
4. f = p.execute(c, 'fp')
5. # result in mbox (spe)
6. r = p.execute(c, 'mbox')

Asynchronous execution:
1. t1 = p.execute(c1, mode='async')
2. t2 = p.execute(c2, mode='async')
3. p.suspend(t2)
4. p.resume(t2)
5. p.stop(t1)
6. p.join(t2)
CorePy and Synthetic Programming

Synthetic programming is a meta-programming technique for synthesizing instruction sequences at run-time from high-level languages.

A Simple Example:

\[ r = ((0 + 31) + 11) \]

1. \( c = \text{InstructionStream()} \)
2. \( \text{ppc.set_active_code}(c) \)
3. \( \text{ppc.addi}(\text{gp_return}, 0, 31) \)
4. \( \text{ppc.addi}(\text{gp return}, \text{gp return}, 11) \)
5. \( p = \text{Processor()} \)
6. \( r = p.\text{execute}(c) \)
7. \( \text{print } r \)
def syn_popc(code):
    spu.set_active_code(code)

    # Reserve four variable registers
    x, count, result, temp = code.acquire_registers(4)

    # 'Load' the input vector x from register 5
    spu.ai(x, 5, 0)

    # Zero count and result
    spu.xor(count, count, count)
    spu.xor(result, result, result)

    # Inline the popc and reduce operations
    spu.cntb(temp, x)
    spu.sumb(temp, temp, 0)
    spu.a(count, count, temp)

    for i in range(4):
        spu.a(result, x, result)
        spu.rotgbyi(x, x, 4)

    # Send the result to the caller
    spu.wrch(result, dma.SPU_WrOutMbox)

code.release_registers(x, count, result, temp)
Calling `syn_popc`

*InstructionStream can be used to collect arbitrary instruction sequences.*

1. `import corepy.arch.spu.platform as env`

2. `# Load the popc SPE program`
3. `code = env.InstructionStream()`
4. `proc = env.Processor()`

5. `# Create the SPE program`
6. `syn_popc(code)`

7. `# Set the input parameters`
8. `params = env.spu_exec.ExecParams()`
9. `params.v3 = [0xDEAD, 0xBEEF, 0x1337, 0xCAFE]`

10. `# Execute the popc program with the parameters`
11. `count = proc.execute(code, 'mbox', params)`
**InstructionStream, Revisited**

*InstructionStream* manages a sequence of instructions, allocates registers, collects object references, and ensures OS ABI (application binary interface) compliance.

Register Allocation:
1. \( ra = c.\text{acquire}_\text{register}() \)
2. \( fa = c.\text{acquire}_\text{register}('fp') \)
3. ... use the registers ...
4. \( c.\text{release}_\text{register}(ra) \)
5. \( c.\text{release}_\text{register}(fa, 'fp') \)

Reference Counting:
1. \( \text{data} = \text{Numeric.ones}(100) \)
2. \( c.\text{add}_\text{storage}(\text{data}) \)
3. ... execute the code ...
4. \( c.\text{release}_\text{storage}() \)

**ABI Compliance:**

- **Prologue:**
  - Save registers
  - Set vector flags

- **... user instructions ...**

- **Epilogue:**
  - Restore registers
ISAs

ISA module exposes each machine instruction as a Python function that generates the proper binary coded machine instruction.

### addx Instruction Example

<table>
<thead>
<tr>
<th>PPC Assembly:</th>
<th>32-bit Binary Layout:</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>(OE = 0, Rc = 0)</td>
</tr>
<tr>
<td>add.</td>
<td>(OE = 0, Rc = 1)</td>
</tr>
<tr>
<td>addo</td>
<td>(OE = 1, Rc = 0)</td>
</tr>
<tr>
<td>addo.</td>
<td>(OE = 1, Rc = 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Ext. Opcode</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>31</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>OE</td>
<td>266</td>
<td>Rc</td>
</tr>
<tr>
<td>add.</td>
<td></td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addo</td>
<td></td>
<td>15</td>
<td>16</td>
<td>20</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addo.</td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rc = 1 sets condition register CR0, OE = 1 overflow register is set

1. `ppc.addx(rd, ra, rb)`  # asm: add D, A, B
2. `ppc.addx(rd, ra, rb, Rc=1)`  # asm: add. D, A, B
3. `ppc.addx(rd, ra, rb, OE=1)`  # asm: addo D, A, B
4. `ppc.addx(rd, ra, rb, Rc=1, OE=1)`  # asm: addo. D, A, B
Variables

CorePy Variables encapsulate a register, backing store, and valid operations for a user-defined data type.

Scalar example:
1. a = SignedWord(11)
2. b = SignedWord(31)
3. c = SignedWord(0, reg=gp_return)
4. c.v = (a + b) * 10
5. --> c = 420

Vector example:
1. a = VecWord([2,3,4,5])
2. b = VecWord([3,3,3,3])
3. c = VecWord(0)
4. c.v = vmin(a, b) * b + 10
5. --> c = [16, 19, 19, 19]
def syn_popc_var(code):
    spu.set_active_code(code)

    x = Word(0)
    count = Word(0)
    result = Word(0)

    # 'Load' the input vector x from register 5
    x.v = spu.ai.ex(5, 0)

    # Inline the popc and reduce operations
    count.v = spu.sumb.ex(spu.cntb.ex(x), 0)

    for i in range(4):
        result.v = result + x
        x.v = spu.rotqbyi.ex(x, 4)

    # Send the result to the caller
    spu.wrch(result, dma.SPU_WrOutMbox)

    code.release_registers(x, count, result)
Iterators enable user-defined loop semantics.

1. # Basic Iteration
2. a = SignedWord(c, 0)

3. for i in syn_iter(c, 5):
4.    for j in syn_iter(c, 5, mode = 'ctr'):
5.       a.v = a + 1

6. proc.execute(c)
7. --> a = 25
Iterator Examples

1. # Array iteration
2. for x in var_iter(c, a): sum.v = sum + x
3. for x in vec_iter(c, a): sum.v = sum + x

4. # Data stream merge
5. for x,y,z,r in zip_iter(c, X,Y,Z,R):
6.   r.v = vmadd(x,y,z)

7. # Auto-parallelization
8. for x in parallel(vec_iter(c, a)): body(x)
9. t1 = proc.execute(c, mode='async', params=[0,2,0])
10. t2 = proc.execute(c, mode='async', params=[1,2,0])

11. # SPE main memory/local stream stream
12. strm = stream_buffer(c, data, buffer size, buffer addr,
13.   double = True, save = True)
14. for buffer in strm:
15.   for x in vec_iter(c, buffer): x.v = x * x
def stream_popc(code):
    # Use the popc and reduce from the previous example
    popc = syn_popc_var()
    x = var.Word(0)
    count = var.Word(0)
    total = var.Word(0)

    # Create a streaming iterator with buffer size < stream size
    stream = stream_buffer(code, stream_addr, stream_size * 4,
                           buffer_size, lsa)

    # Create an array descriptor for the LS buffer
    ls data = memory desc('I', lsa, buffer size / 4)

    # Count the population in the stream
    for buffer in stream:
        for x in spu vec iter(code, ls data, addr reg = buffer):
            popc.popc(count, x)
            popc.reduce_word(total, count)

    # Send the result to the caller
    spu wrch(total, dma.SPU_WrOutMbox)

    return
Debugging

CorePy programs can be **debugged using** `gdb` or the CorePy SPU debugger. Debug printouts provide memory, register, and source details.

```python
8. def sum(code):
9.
10. a = ppcvar.UnsignedWord(0)
11.
12. for i in syn_range(code, 0, 100, 2):
13.     a.v = a + i
14.
15. return
```

```
0x003894E0  0 addi(r30,0,0)     [debug.py:sum: 9]   a = ppcvar.UnsignedWord(0)
0x003894E4  1 addi(r29,0,0)     [debug.py:sum: 11]   for i in syn_range(code, 0,100,2):
0x003894E8  2 addi(r28,0,100)    return_var("a")
0x003894EC  3 ori(0,0,0)          "" ""
0x003894F0  4 addx(r30,<r30>,<r29>) [debug.py:sum: 12]   a.v = a + i
0x003894F4  5 addi(r29,r29,2)    [debug.py:sum: 11]   for i in syn_range(code, 0,100,2):
0x003894F8  6 cmp_(0,0,r29,r28)    "" ""
0x00389500  7 "" "" "" "" ""
```
Light-weight SPU Debugger

CorePy, combined with the unique execution model of the Cell SPUs, makes it possible to implement a simple, interactive SPU debugger entirely in Python.

Example:

```python
# Create the synthetic program
code = InstructionStream()
spu.set_active_code(code)

spu.ai(127, 0, 1)
spu.ai(126, 0, 2)
spu.ai(125, 0, 3)
spu.brnz(125, 2)
spu.ai(124, 0, 4)
spu.ai(123, 0, 5)

# Execute it with a debug processor
proc = DebugProcessor()
r = proc.execute(code)

while r is not None:
    regs = proc.get_regs()
    print [reg[0] for reg in regs[123:]]
    r = proc.nexti()

# Output
---> [1,0,0,0,0]
---> [1,2,0,0,0]
---> [1,2,3,0,0]
---> [1,2,3,0,5]
```
The **Interactive SPU** module provides a graphical or command line interface to a running SPU.
Other Applications

Chemical fingerprint comparison
Rat brain neural analysis
Cell BLAST prototype
SPU big num library
PS3 framebuffer support
VMX particle system
Scalable PPC dgemm
Interactive SPE
## CorePy Status

<table>
<thead>
<tr>
<th></th>
<th>SPE</th>
<th>PPE</th>
<th>VMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>🔄</td>
<td>🔄</td>
<td>(☆)</td>
</tr>
<tr>
<td>InstructionStream</td>
<td>🔄</td>
<td>🔄</td>
<td>(☆)</td>
</tr>
<tr>
<td>NativeInstStream</td>
<td>🔄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debug Print</td>
<td></td>
<td></td>
<td>(☆)</td>
</tr>
<tr>
<td>Interactive Debug</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>CorePy - Beta</td>
<td>Linux OS X - Beta</td>
<td>Linux OS X - Alpha</td>
</tr>
</tbody>
</table>

- **Availability**
  - Source distribution
  - Evaluation license

- **Tested Platforms**
  - IBM Cell Blades
  - PS3
  - Apple G4/G5 Macs

- **Caveat**
  - libspe 1.x
Conclusion

CorePy provides a framework for rapid development of high-performance applications on the Cell/B.E.

CorePy Benefits:

- Control C/C++/etc SPU programs from Python
- Develop assembly-level code using an intuitive API
- Design and debug new SPU SIMD algorithms interactively
Thank You!

Special Thanks To:

IBM Cell Ecosystem Team, especially:
Hema Reddy, Gordon Ellison, Jennifer Turner

The Lilly Foundation

http://www.corepy.org
chemuell@cs.indiana.edu

“CorePy makes assembly fun again!”
-Alex Breuer