Synthetic Programming on the Cell BE

Christopher Mueller, Benjamin Martin and Andrew Lumsdaine

October 25, 2006

UT/IBM Cell Workshop
Modern PowerPC Processors

PowerPC 970FX
- SIMD/Vector Unit
- Dual Floating Point Unit
- Instruction Sequencing Unit
- Dual Fixed Point Units
- Instruction Fetch Unit
- Dual Load Store Units
- L1 Instruction Cache
- L1 Data Cache
- Storage Subsystem
- L2 Cache

IBM Cell BE
- L2 (512KB)
- Test & Debug Logic
- Power Processor Element
- Element Interconnect Bus
- I/O Controller
- Hamms FlexIO™
Programming Modern Processors

A Python HPC Stack

- Processor
- Linker
- Optimizer
- Compiler
- Language
- Extension Library
- Inliner
- Wrapper Generator
- Python

Streamlined (high productivity)

Skillset/ Toolchain

Complex

No man’s land

A Python HPC Stack

- Processor
- Linker
- Optimizer
- Compiler
- Language
- Extension Library
- Inliner
- Wrapper Generator
- Python

Streamlined (high productivity)

Skillset/ Toolchain

Complex

No man’s land
What if..?
The Synthetic Programming Environment (SPE) is a user-level library for synthesizing high-performance computational kernels at runtime.

1. `c = InstructionStream()`
2. `c.add(ppc.addi(gp_return, 0, 31))`
3. `c.add(ppc.addi(gp_return, gp_return, 11))`
4. `p = Processor()`
5. `r = p.execute(c)`
6. `print r`
7. `--> 42`
Synthetic Programming

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

Terminology:

- **Synthetic programs**: complete instruction streams
- **Synthetic components**: meta-programming functions and objects that synthesize partial instruction sequences
- **Synthetic ***: modifier to denote synthesized code, e.g. synthetic loop
Example: add/reduce

Synthetic:
1. `# r_* are processor registers, e.g. r_sum = 3`
2. `c = InstructionStream()`
3. `n = 10000000`
4. `a = array(range(n))`
5. `# Load a pointer to the array`
6. `c.load_word(r_addr, addr(a))`
7. `# Set the counter`
8. `c.load_word(r_temp, n)`
9. `c.add(ppc.mtctr(r_temp))`
10. `# Zero the sum`
11. `c.add(ppc.addi(r_sum, 0, 0))`
12. `# Set a loop label and load the next value`
13. `start = c.add(ppc.lwz(r_current, r_addr, 0))`
14. `# Update the sum`
15. `c.add(ppc.add(r_sum, r_sum, r_current))`
16. `# Increment the pointer`
17. `c.add(ppc.addi(r_addr, r_addr, 4))`
18. `# Decrement the counter and loop`
19. `next = code.size() + 1`
20. `c.add(ppc.bdnz(-((next - start) * 4)))`
21. `result = proc.execute(c)`

Numeric:
1. `n = 10000000`
2. `a = Numeric.arange(n)`
3. `sum = Numeric.add.reduce(a)`

Python:
1. `n = 10000000`
2. `a = array(range(n))`
3. `sum = 0`
4. `for i in a:`
5. `s += i`

<table>
<thead>
<tr>
<th>Test</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>0.019</td>
</tr>
<tr>
<td>Numeric</td>
<td>0.054</td>
</tr>
<tr>
<td>Python</td>
<td>3.64</td>
</tr>
</tbody>
</table>

2.5 MHz G5, 3 GB, timed code gen and reduce operations only
Synthetic code is executed on the PPU using a thin native interface. ABI compliance is managed in Python.

Parameters and return values:
1. # Pass two parameters
2. v = Params()
3. v.p1 = 31
4. v.p2 = 11
5. p.execute(c, params=v)

6. # result in gp_return
7. r = p.execute(c)

8. # result in fp_return
9. f = p.execute(c, ‘fp’)
Synthetic code is generated on the PPU and executed on the SPU using a native interface and an SPU bootstrap program.

Single SPU, blocking execution:
1. # Execute and stop
2. c.add(spu.stop(0x7))
3. # result from stop
4. r = p.execute(c)
5. --> r = 0x7

8 SPUs, non-blocking execution:
1. ids = p.execute(c, mode = 'async', n_spus = 8)
2. for id in ids:
3.   p.join(id)
**Synthetic Variables**

*Synthetic variables* encapsulate a register, backing store, and valid operations for a processor data type.

**Scalar variables (PPU):**

1. `a = var(0)`
2. `b = var(1.2)`
3. `c = var(4, reg=gp_return)`
4. `b.store()`
5. `... execute code ...`
6. `print b.value`
7. `--> 1.2`

**Vector variables (PPU/SPU):**

1. `a = vector(1)`
2. `b = vector([1,2,3,4])`
3. `AltiVec.vadd(a.reg, a.reg, b.reg)`
4. `... execute code ...`
5. `6. ... execute code ...`
6. `7. --> a = [2,3,4,5]`
**Synthetic Expressions** use synthetic variables and Python operators to generate instruction sequences for arbitrary expressions.

**Scalar expressions (PPU):**

1. \( a = \text{var}(11) \)
2. \( b = \text{var}(31) \)
3. \( c = \text{var}(0, \text{reg}=\text{gp\_return}) \)
4. \( c = (a + b) \times 10 \)
5. \( \rightarrow c = 420 \)

**Vector expressions (PPU/SPU):**

1. \( a = \text{vector}([2,3,4,5]) \)
2. \( b = \text{vector}([3,3,3,3]) \)
3. \( c = \text{vector}(0) \)
4. \( c = \text{vmin}(a, b) \times b + 10 \)
5. \( \rightarrow c = [16, 19, 19, 19] \)
6. \( d = \text{selb}((a < b), a, b) \)
7. \( \rightarrow d = [2,3,3,3] \)
**Synthetic Iterators** synthesize code to manage loops on both the PPU and SPU.

1. # Basic Iteration (PPU/SPU)  
2. a = var(c, 0)  
3. for i in syn_iter(c, 5):  
4.   for j in syn_iter(c, 5):  
5.     a.v = a + i + j  
6. proc.execute(c)  
7. --> a = 100
1. for i in syn_iter(c, 20):
2.   sum.v = sum + i
Array Iterators

Array iterators iterate over values in an array, one element at a time or in vector-size steps.

1. \( a = \text{arange}(32) \)

2. 
   
   ```
   # Serial add/reduce (PPU)
   for i in var_iter(c, a):
   sum.v = sum + i
   ```

3. 

5. 
   
   ```
   # SIMD add/reduce (PPU/SPU)
   for i in vec_iter(c, a):
   sum.v = sum + i
   ```
Pythonic Iterators

`syn_range` and `zip_iter` duplicate the functionality of Python’s `range` and `zip`.

1. `for i in syn_range(c, 10, 20, 2):`
2. `a.v = a + i`
3. `--> a = 70`

1. `# X,Y,Z,R are vec_iters`
2. `for x, y, z, r in zip_iter(c, X, Y, Z, R):`
3. `r = vmadd(x, y, z)`
4. `--> r[i] = x[i]*z[i] + y[i]`
High-Performance Iterators

High-Performance iterators modify existing iterators for optimized code generation.

1. # Unroll the loop 3 times
2. for x,y,z,r in unroll(zip_iter(c, X,Y,Z,R), 3):
3.   r = vmadd(x,y,z)

1. # Divide the data for parallel execution
2. c = ParallelInstructionStream()

3. for x,y,z,r in parallel(zip_iter(c, X,Y,Z,R)):
4.   r = vmadd(x,y,z)

5. t1 = proc.execute(c, mode='async', params=[0,2,0])
6. t2 = proc.execute(c, mode='async', params=[1,2,0])
Stream Buffer Iterators

*stream_buffer iterators* move data between main memory and the SPU local store.

1. `stream = stream_buffer(c, data, buffer_size, buffer_addr, double = True, save = True)`
2. `stream = parallel(stream)`
3. `for buffer in stream:`
4. `for x in vec_iter(c, buffer):`
5. `x.v = x * x`
6. `proc.execute(c, n_spus = 4)`
Communication

All libspe functions are available to the host program.

```python
# Send a message to the PPU
spu_write_out_mbox(code, 0xDEADBEAF)

# Get a message from the PPU
reg = spu_read_in_mbox(code)

# And send it back
code.add(spu.wrch(SPU_WrOutMbox, reg))

# Execute the synthetic program
spe_id = proc.execute(code, mode='async')
write_in_mbox(spe_id, 0x88CAFE)

while stat_out_mbox(spe_id) != 0:
    msg = read_out_mbox(spe_id)

proc.join(spe_id)
```
Example: BLAST

Step 1: Find all high-scoring words in the query

Step 2: For each database subject, find all instances of the high scoring words.

Step 3: Extend the matches in each direction to generate high scoring pairs.
Synthetic BLAST on the Cell BE
Conclusion

- Synthetic Programming
  - Rapid development at the hardware level
  - Python libraries available for admin tasks
  - Meta-programming enables new software architectures (see also C++ Boost libraries)

- …on the Cell BE
  - Effective model for managing SPU execution
  - More fun than cross-compiling

- Wish list
  - SPU vector lookup
  - mfc_gets safely exposed on PPU
Thank you!

www.synthetic-programming.org

chemuell@cs.indiana.edu
Multi-Core SPE: Cell BE (1)

The cell BE contains two general purpose PPC cores and eight special purpose Synergistic Processing Element* cores.

Challenges and Opportunities:

- Three ISAs - PowerPC, SPU, AltiVec
- Explicit control of data transfers between main memory and SPUs
- In-order execution on SPUs
- Limited access to Cell BE blades
- Linux port designed for C-based tool chain

Programmer’s view of the Cell BE

*SPE has a specific meaning in Cell BE jargon. In this discussion, SPE will refer to the Synthetic Programming Environment and SPU the Cell BE’s Synergistic Processing Unit.
Multi-Core SPE: Cell BE (2)

The Synthetic Programming Environment for the Cell BE includes support for **SPU code synthesis** and **execution**, using both the **SPU ISA** and the higher-level **expression** and **iterator** libraries.

Double Buffered Stream Operation:

1. \( c = \text{SPU.InstructionStream}() \)
2. \( s = \text{stream_buffer}( \ldots ) \)
3. \( \text{stream} = \text{parallel}(s) \)
4. \( \text{md} = \text{memory_desc}( \ldots ) \)
5. \( \text{for} \ \text{buffer} \ \text{in} \ \text{stream}: \)
6. \( \text{for} \ \text{x in} \ \text{spu_vec_iter}(c, \text{md}): \)
7. \( \quad x.v = x + x \)
8. \( \text{r} = \text{proc.execute}(c, n\_spus = 8) \)

Asynchronous execution and Cell BE communication mechanisms enable much more complex and powerful system architectures.