
And now for something completely different...

CorePy: High-Productivity Cell/B.E. Programming

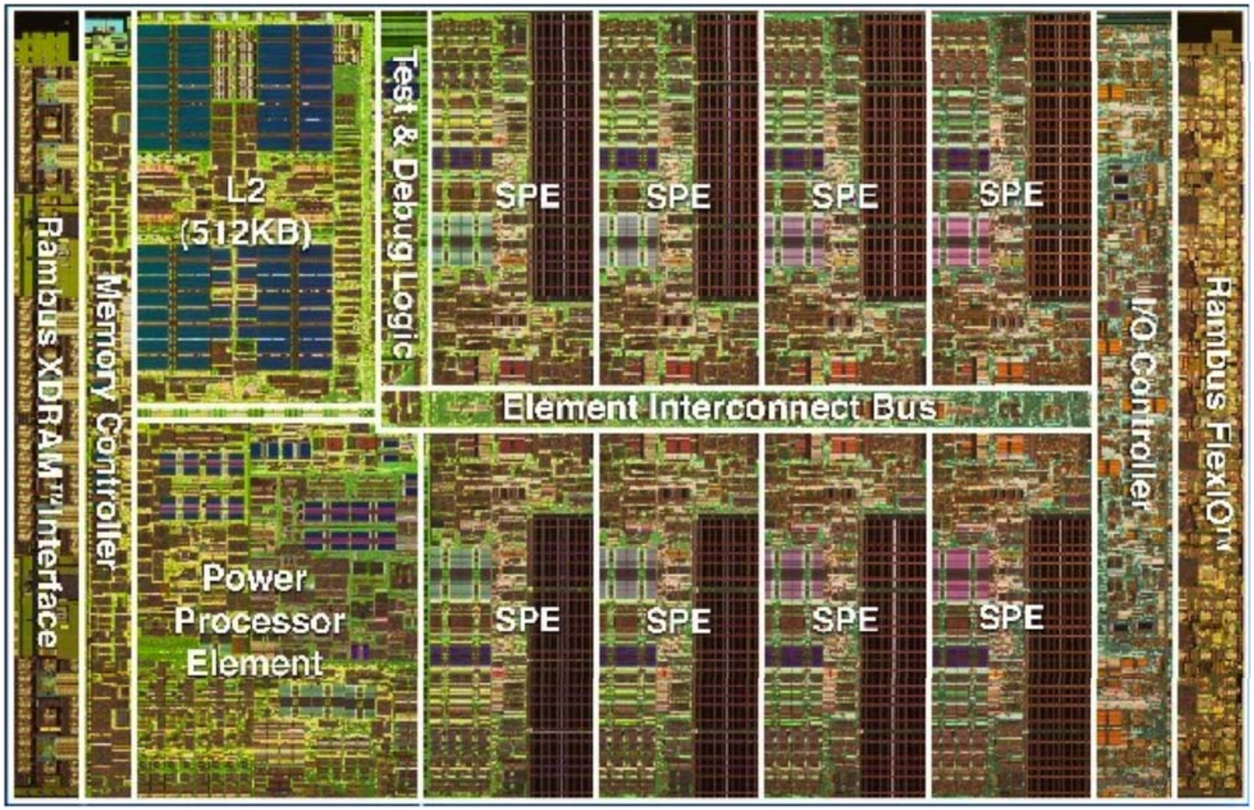
Chris Mueller, Ben Martin, and Andrew Lumsdaine

Indiana University

June 19, 2007

Georgia Tech, Sony/Toshiba/IBM Workshop on Software and Applications for the Cell/B.E. processor

Cell/B.E.



Rambus XDRAM™ Interface
Memory Controller
L2
(512KB)
Power
Processor
Element
Test & Debug Logic
SPE
SPE
SPE
SPE
Element Interconnect Bus
SPE
SPE
SPE
SPE
I/O Controller
Rambus FlexIO™

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!*

*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…

(somewhere between machine code and assembly)

The Approach

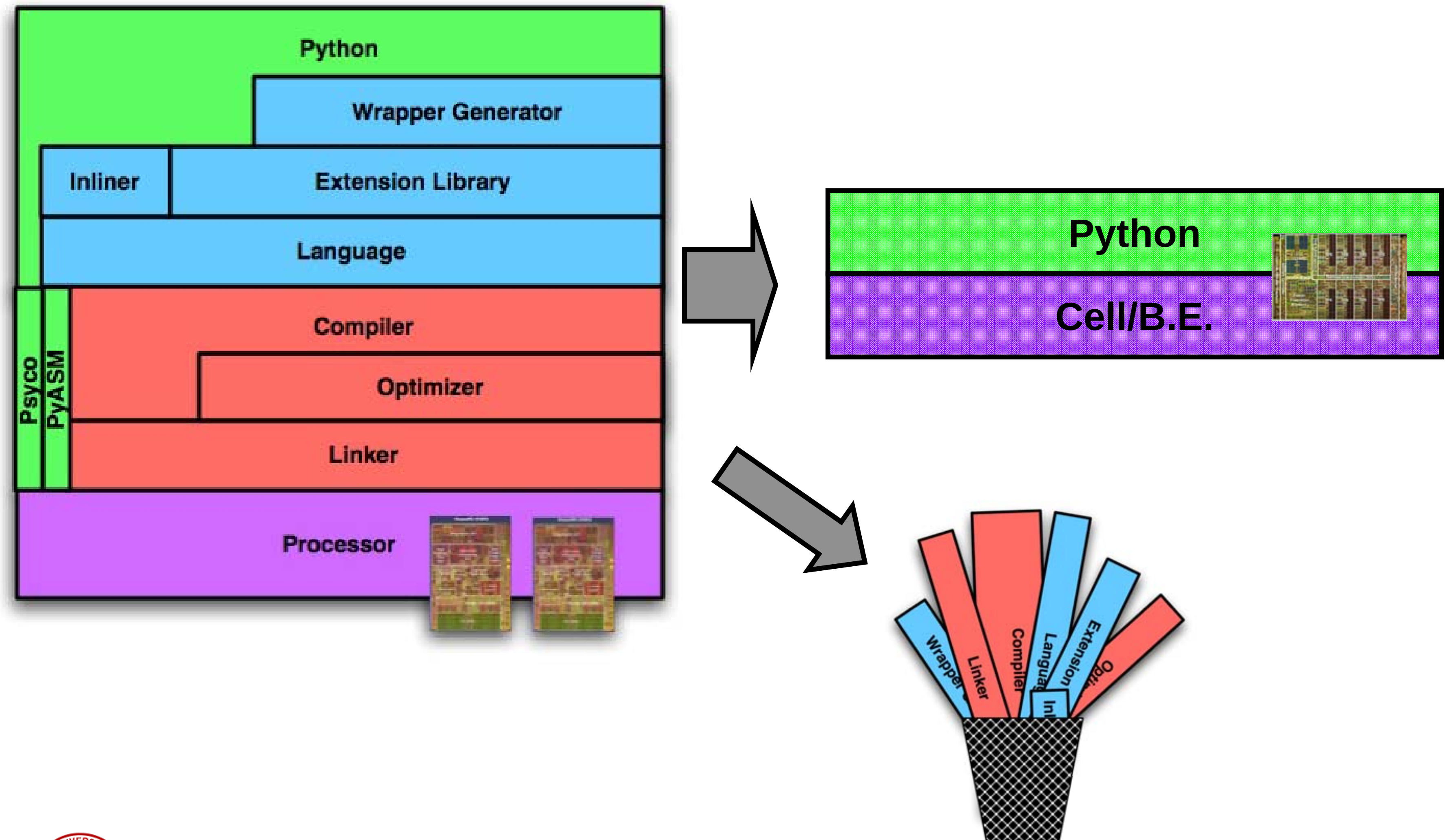
Take a modern programming technique…

…provide direct access to the hardware…

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

(power to the people?)

The Approach, Illustrated

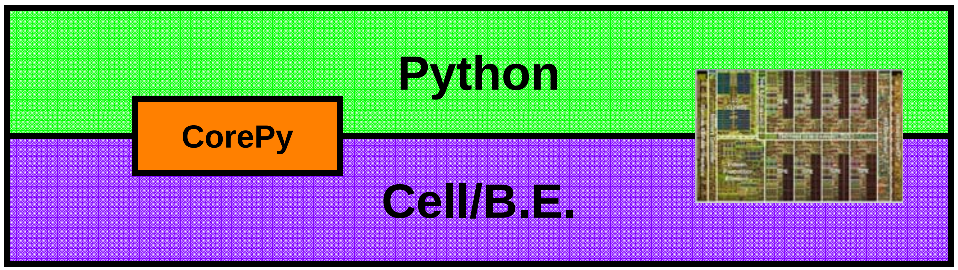


Huh?!? Python for Cell Programming?

- 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

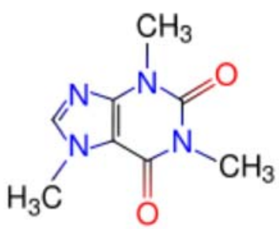
Instruction Set Architecture (ISA)	PowerPC, VMX, and SPU instruction interface
Variables	Library-defined “primitive” types
Iterators	Loop generation and optimization

Example: Population Count

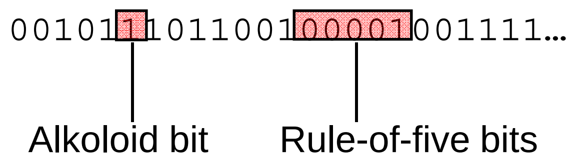
Population count (popc) counts the number of '1' bits in a bit vector. It is used in many data clustering and machine-learning algorithms to compare feature vectors.

```
v = {0b0101010111101101, 0b1100110000000001,
     0b0000001000100001, 0b1111111111000001}
c = popc(v)
print c
-> 29
```

Structure



Bit-vector Representation



Metrics

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

$$D_{Jaccard} = \frac{c}{a+b+c}$$

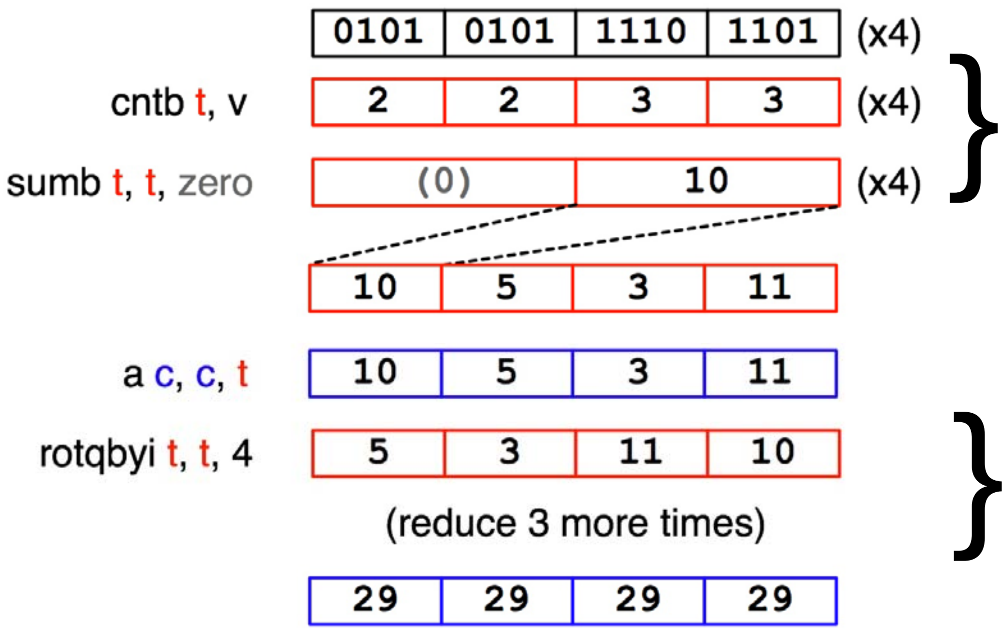
x: 101011101
y: 110011010
cbadccaba

$D_{Rand} = .444, D_{Jaccard} = .375$

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



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

```c
#include "corepy.h"

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

  // Initialize CorePy run time
  spu_ready();

  count = (vector unsigned int)spu_cntb((vector unsigned char)x);
  count = (vector unsigned int)spu sumb((vector unsigned char)count,
                                        (vector unsigned char)0);

  for(i; i < 4; i++) {
    result = result + x;
    x = si_rotqbyi(x, 4);
  }

  // Report the result using the out mailbox
  spu write out mbox(spu extract(result, 0));

  return SPU_SUCCESS;
}
```

Calling `spu_popc` Using CorePy

The Processor object executes SPE programs and retrieves return values.

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

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

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

# 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
c = env.NativeInstructionStream("spu_popc")
```

***Processor** manages execution and return values.*

Return values:

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

# result in fp_return
f = p.execute(c, ‘fp’)

# result in mbox (spe)
r = p.execute(c, ‘mbox’)
```

Asynchronous execution:

```
t1 = p.execute(c1, mode=‘async’)
t2 = p.execute(c2, mode=‘async’)

p.suspend(t2)
p.resume(t2)

p.stop(t1)
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:

$$r = ((0 + 31) + 11)$$

```
c = InstructionStream()
ppc.set_active_code(c)

ppc.addi(gp_return, 0, 31)
ppc.addi(gp return, gp return, 11)

p = Processor()
r = p.execute(c)


print r
```

Synthetic Population Count

```
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.rotqbyi(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.

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

# Load the popc SPE program
code = env.InstructionStream()
proc = env.Processor()

# Create the SPE program
syn_popc(code)

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

# Execute the popc program with the parameters
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.*

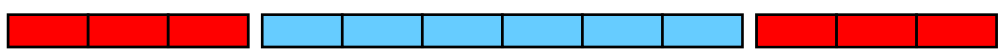
Register Allocation:

```
ra = c.acquire_register()
fa = c.acquire_register('fp')
 … use the registers …
c.release_register(ra)
c.release_register(fa, 'fp')
```

Reference Counting:

```
data = Numeric.ones(100)
c.add_storage(data)
  … execute the code …
c.release_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

PPC Assembly:

add	D, A, B	(OE = 0, Rc = 0)
add.	D, A, B	(OE = 0, Rc = 1)
addo	D, A, B	(OE = 1, Rc = 0)
addo.	D, A, B	(OE = 1, Rc = 1)

32-bit Binary Layout:

Opcode	Field	Field	Field	Field	Ext. Opcode	Field
31	D	A	B	OE	266	Rc
0 – 5	6 – 10	11 – 15	16 – 20	21	22 – 30	31

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

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

```
a = SignedWord(11)
b = SignedWord(31)
c = SignedWord(0, reg=gp_return)

c.v = (a + b) * 10
--> c = 420
```

Vector example:

```
a = VecWord([2,3,4,5])
b = VecWord([3,3,3,3])
c = VecWord(0)

c.v = vmin(a, b) * b + 10
--> c = [16, 19, 19, 19]
```

Synthetic Population Count, Take 2

```
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

Iterators enable user-defined loop semantics.

```
# Basic Iteration
a = SignedWord(c, 0)

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

proc.execute(c)
--> a = 25
```

Iterator Examples

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

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

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

# SPE main memory/local stream stream
strm = stream_buffer(c, data, buffer size, buffer addr,
                            double = True, save = True)
for buffer in strm:
  for x in vec_iter(c, buffer): x.v = x * x
```

CorePy Population Count, Take 3

```
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.*

```
def sum(code):


  a = ppcvar.UnsignedWord(0)



  for i in syn range(code, 0, 100, 2):

    a.v = a + i

  return_var(a)


  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)          ""  ""
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)        ""  ""
```

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:

```
# 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)
```

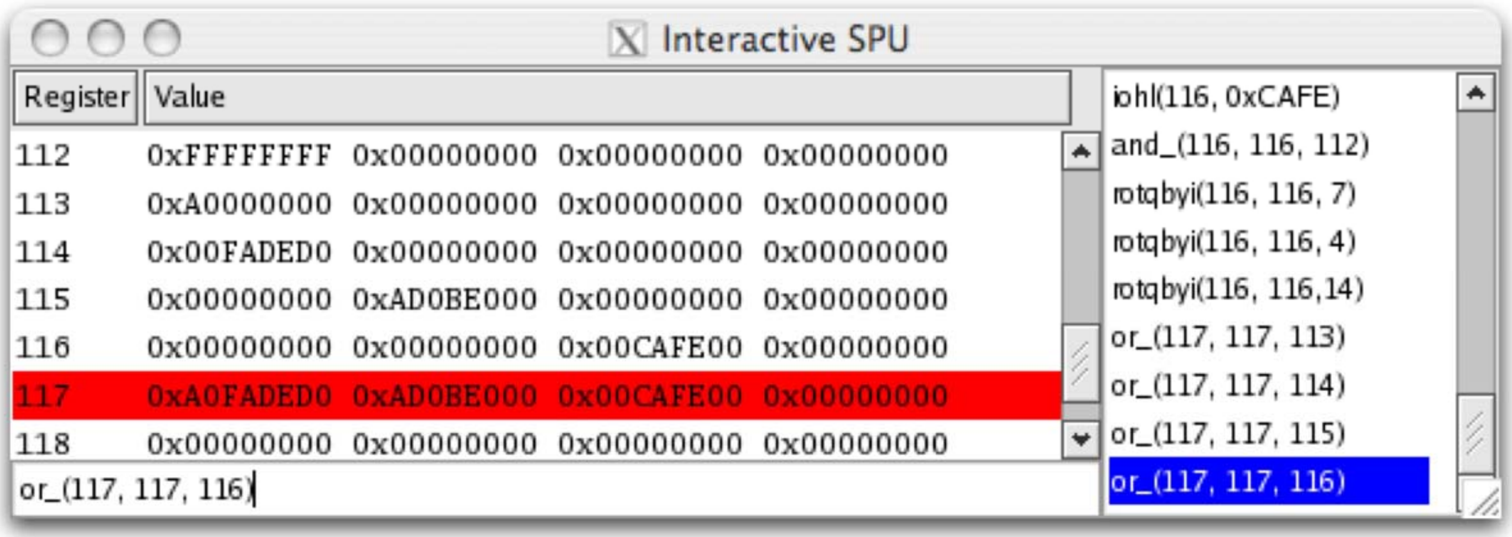
```
# Execute it with a debug processor
proc = DebugProcessor()
r = proc.execute(code)

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]
```

Interactive SPU

*The **Interactive SPU** module provides a graphical or command line interface to a running SPU.*

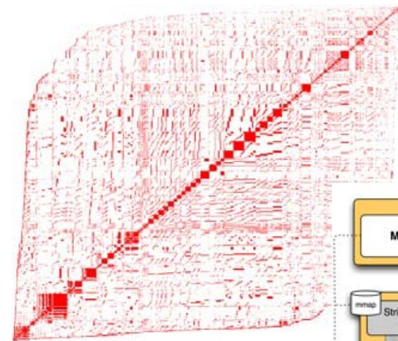


pervasivetechnologylabs
AT INDIANA UNIVERSITY

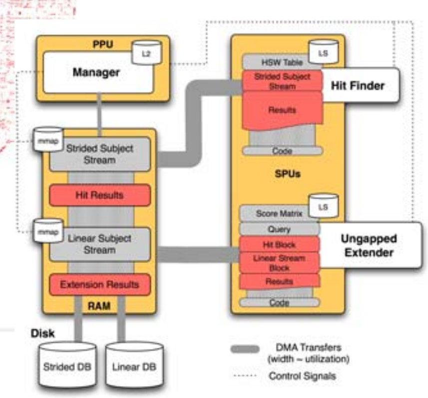
Other Applications

Chemical fingerprint comparison

Rat brain neural analysis



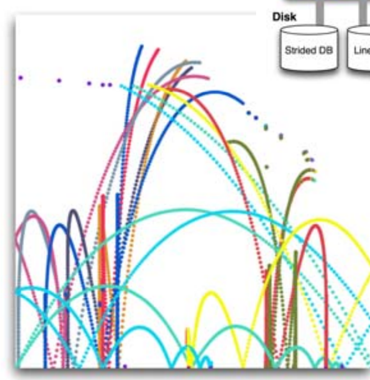
Cell BLAST prototype



SPU big num library

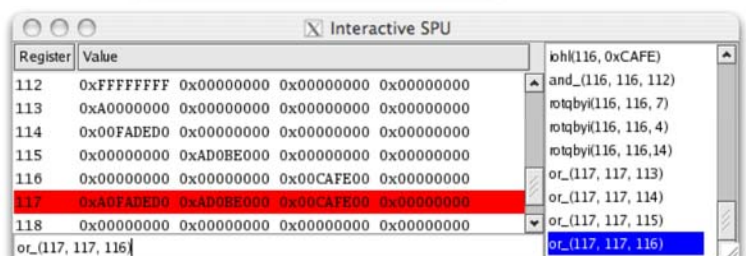
PS3 framebuffer support

VMX particle system



Scalable PPC dgemm

Interactive SPE



AT INDIANA UNIVERSITY

CorePy Status

	SPE	PPE	VMX
Processor	Beta	Beta	(Beta)
InstructionStream	Beta	Beta	(Beta)
NativeInstStream	Beta		
Variables	Alpha	Alpha	Alpha
Iterators	Alpha	Alpha	Alpha
Debug Print	Beta	Beta	(Beta)
Interactive Debug	Alpha		
Operating System	CorePy Linux	Linux OS X	Linux OS X

- Beta - Alpha

- 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 assemblyfun again!"
-Alex Breuer