Synthetic Programming on the Cell BE

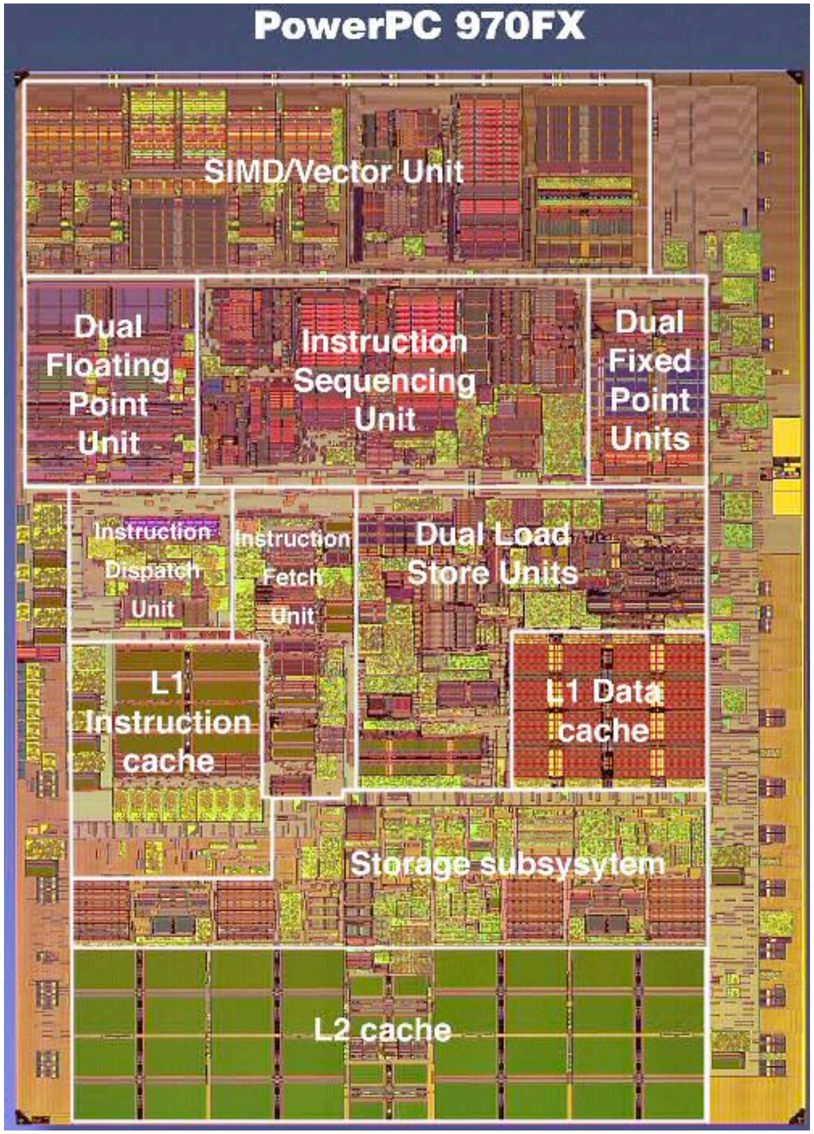
Christopher Mueller, Benjamin Martin and Andrew Lumsdaine

October 25, 2006

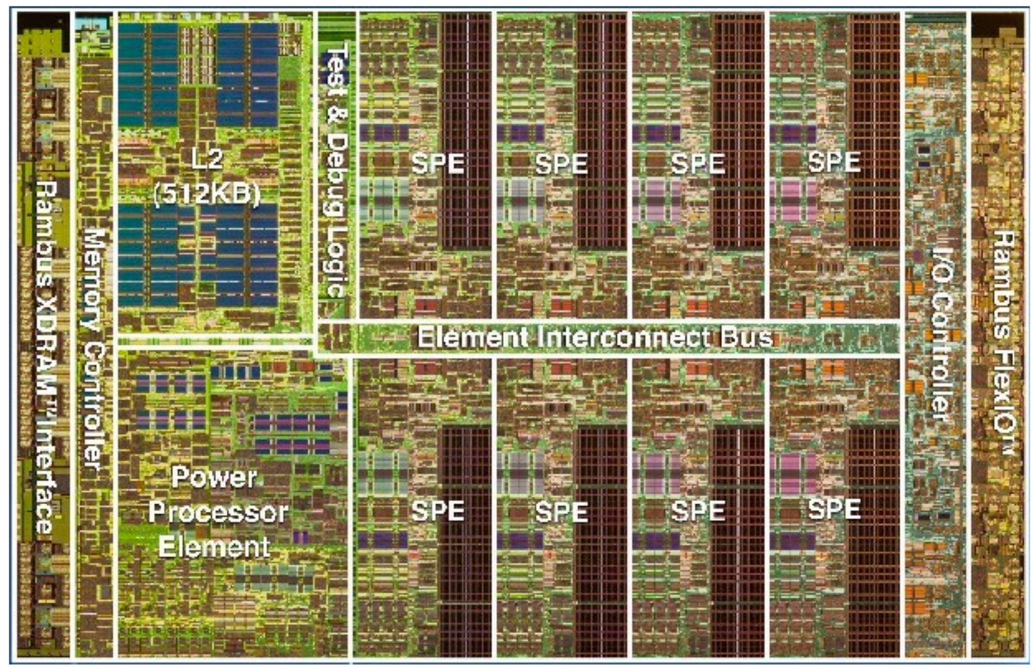
UT/IBM Cell Workshop

pervasivetechnologylabs
AT INDIANA UNIVERSITY

Modern PowerPC Processors

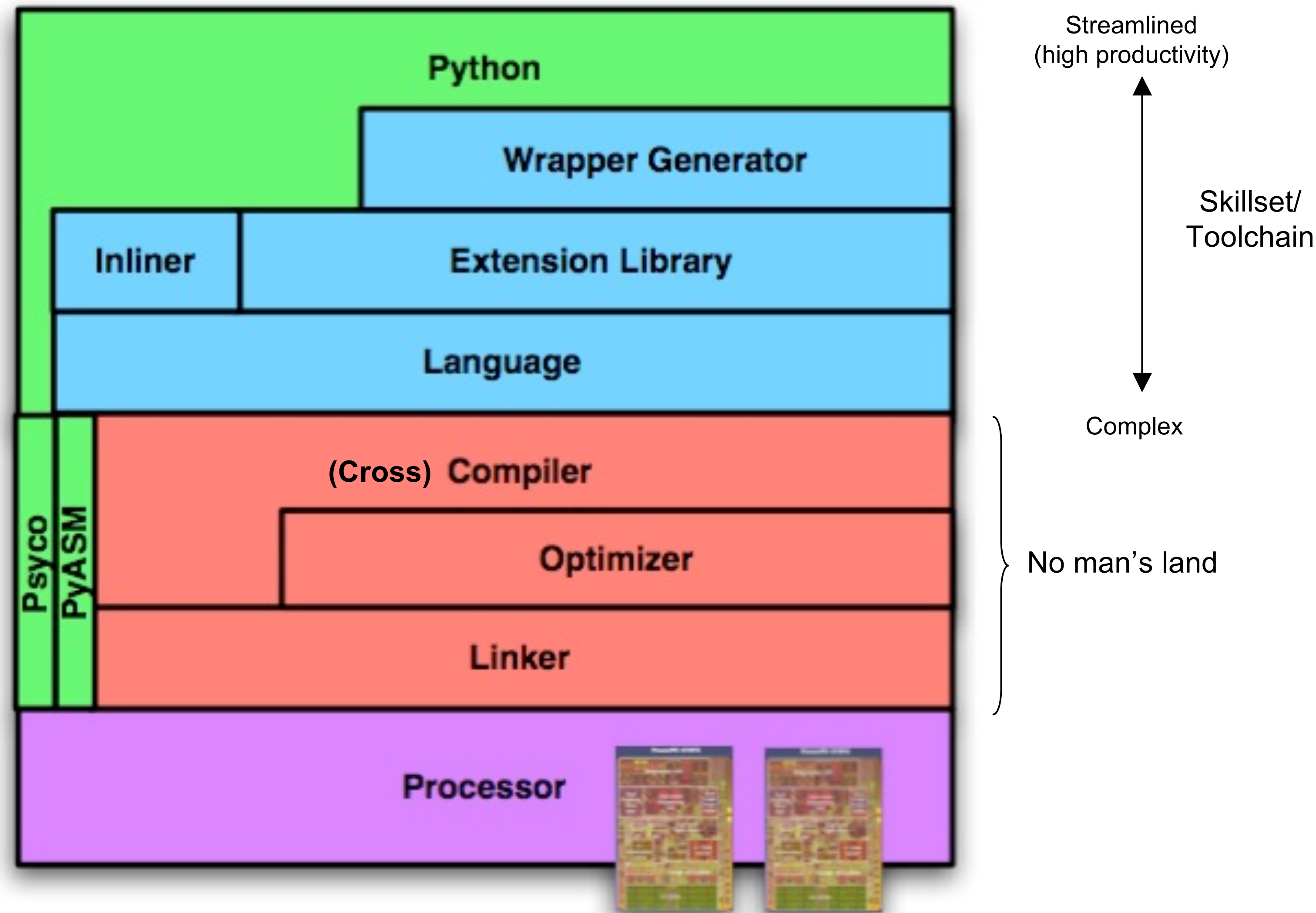


IBM Cell BE

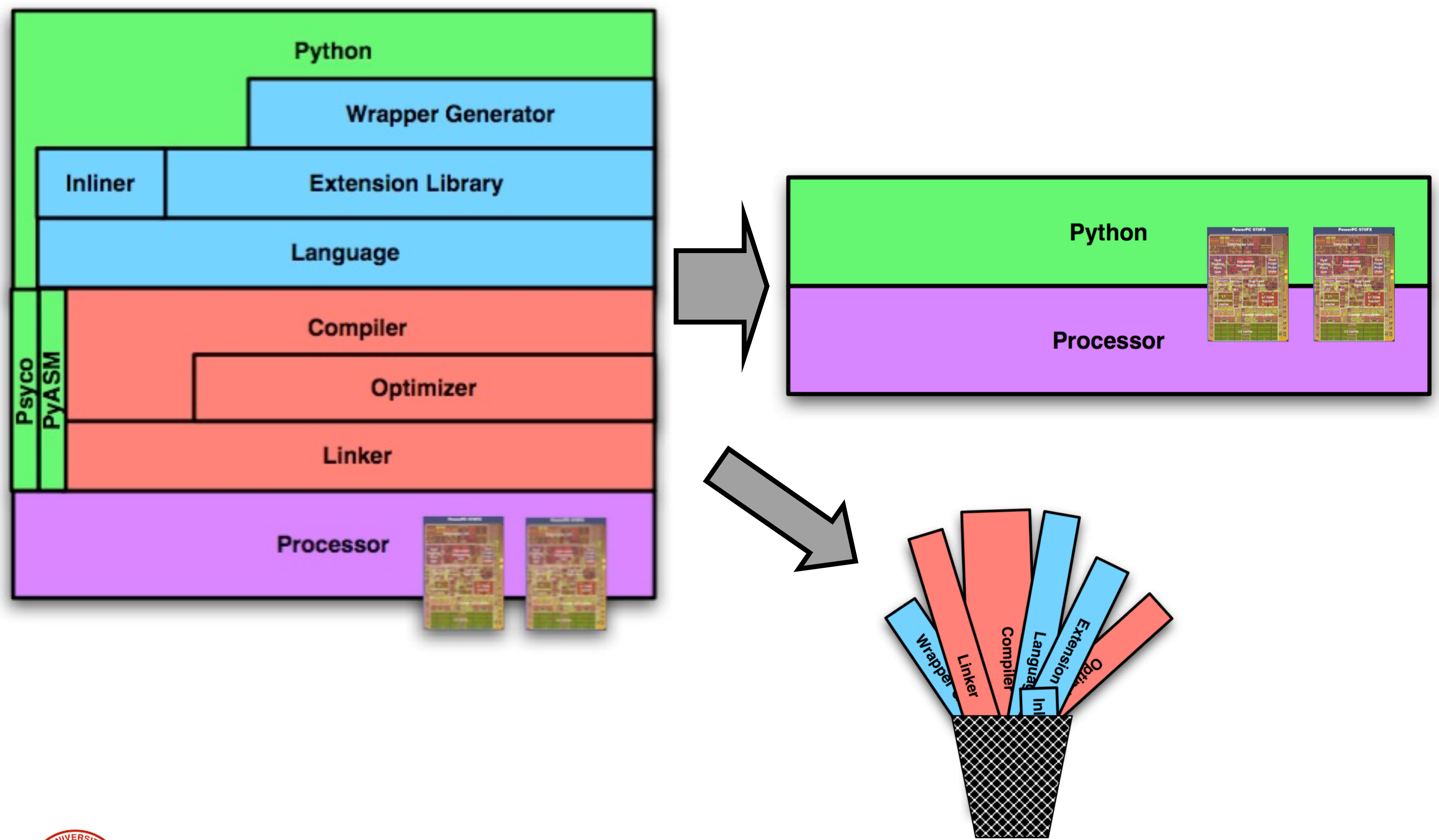


Programming Modern Processors

A Python HPC Stack



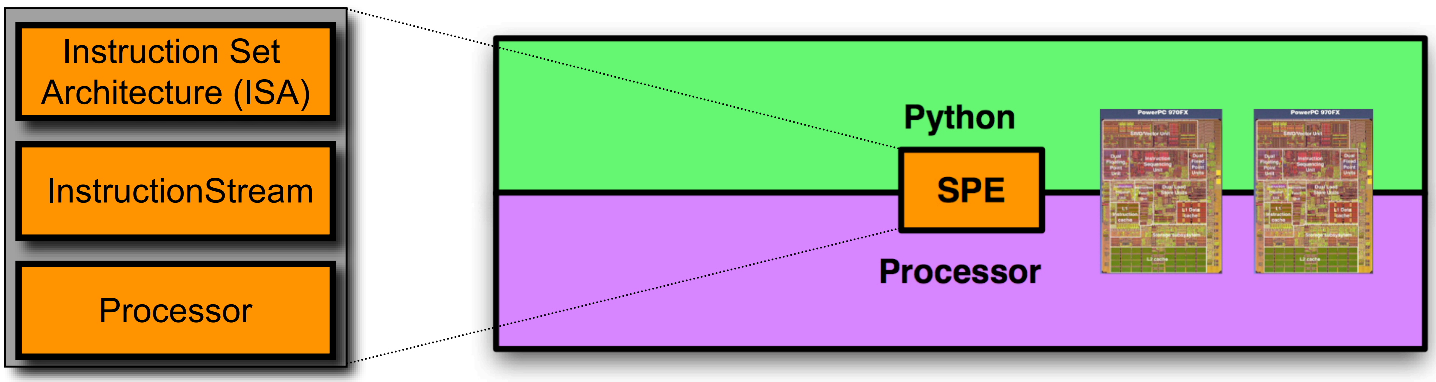
What if..?



Python
Wrapper Generator
Inliner
Extension Library
Language
Psyco
PyASM
Compiler
Optimizer
Linker
Processor
Python
Processor
Wrapper
Linker
Compiler
Extension

Synthetic Programming Environment

The Synthetic Programming Environment (SPE) is a user-level library for synthesizing high-performance computational kernels at runtime.



r = ((0 + 31) + 11)

```
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:

```
# r_* are processor registers, e.g. r_sum = 3
c = InstructionStream()
n = 10000000
a = array(range(n))

# Load a pointer to the array
c.load_word(r_addr, addr(a))

# Set the counter
c.load_word(r_temp, n))
c.add(ppc.mtctr(r_temp))

# Zero the sum
c.add(ppc.addi(r_sum, 0, 0))

# Set a loop label and load the next value
start = c.add(ppc.lwz(r_current, r_addr, 0))

# Update the sum
c.add(ppc.add(r_sum, r_sum, r_current))

# Increment the pointer
c.add(ppc.addi(r_addr, r_addr, 4))

# Decrement the counter and loop
next = code.size() + 1
c.add(ppc.bdnz(-((next - start) * 4)))

result = proc.execute(c)
```

Numeric:

```
n = 10000000
a = Numeric.arange(n)
sum = Numeric.add.reduce(a)
```

Python:

```
n = 10000000
a = array(range(n))
sum = 0

for i in a:
  s += i
```

Test	Time (s)
Synthetic	0.019
Numeric	0.054
Python	3.64

2.5 MHz G5, 3 GB, timed code gen and reduce operations only

Execution - PPU

Synthetic code is executed on the PPU using a thin native interface. ABI compliance is managed in Python.

Parameters and return values:

```
# Pass two parameters
v = Params()
v.p1 = 31
v.p2 = 11
p.execute(c, params=v)

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

# result in fp_return
f = p.execute(c, 'fp')
```

Asynchronous execution:

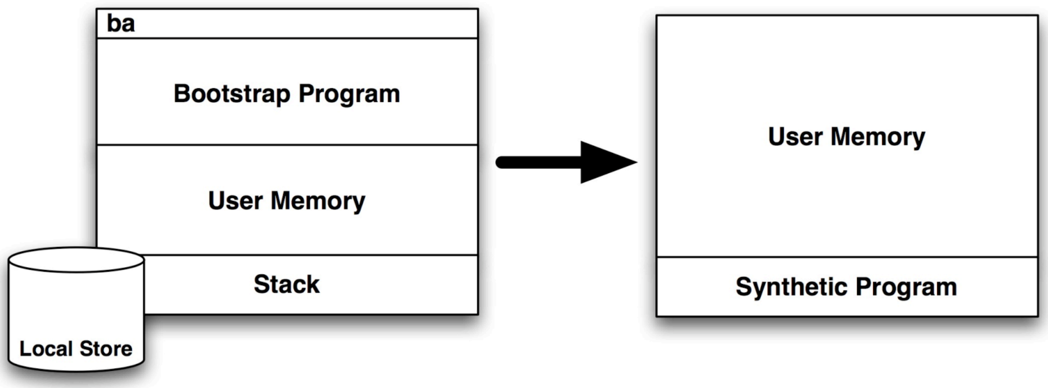
```
t1 = p.execute(c1, mode='async')
t2 = p.execute(c2, mode='async')

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

p.stop(t1)
p.join(t2)
```

Execution - SPU

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:

```
# Execute and stop
c.add(spu.stop(0x7))

# result from stop
r = p.execute(c)
--> r = 0x7
```

8 SPUs, non-blocking execution:

```
ids = p.execute(c,
          mode = 'async',
          n_spus = 8)

for id in ids:
  p.join(id)
```

Synthetic Variables

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

Scalar variables (PPU):

```
a = var(0)
b = var(1.2)
c = var(4, reg=gp_return)

b.store()
  … execute code …

print b.value
--> 1.2
```

Vector variables (PPU/SPU):

```
a = vector(1)
b = vector([1,2,3,4])

AltiVec.vadd(a.reg,
             a.reg,
             b.reg)

… execute code …
--> a = [2,3,4,5]
```

Synthetic Expressions

Synthetic expressions use synthetic variables and Python operators to generate instruction sequences for arbitrary expressions.

Scalar expressions (PPU):

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

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

Vector expressions (PPU/SPU):

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

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

d = selb((a < b), a, b)
--> d = [2,3,3,3]
```

Synthetic Iterators

Synthetic iterators synthesize code to manage loops on both the PPU and SPU.

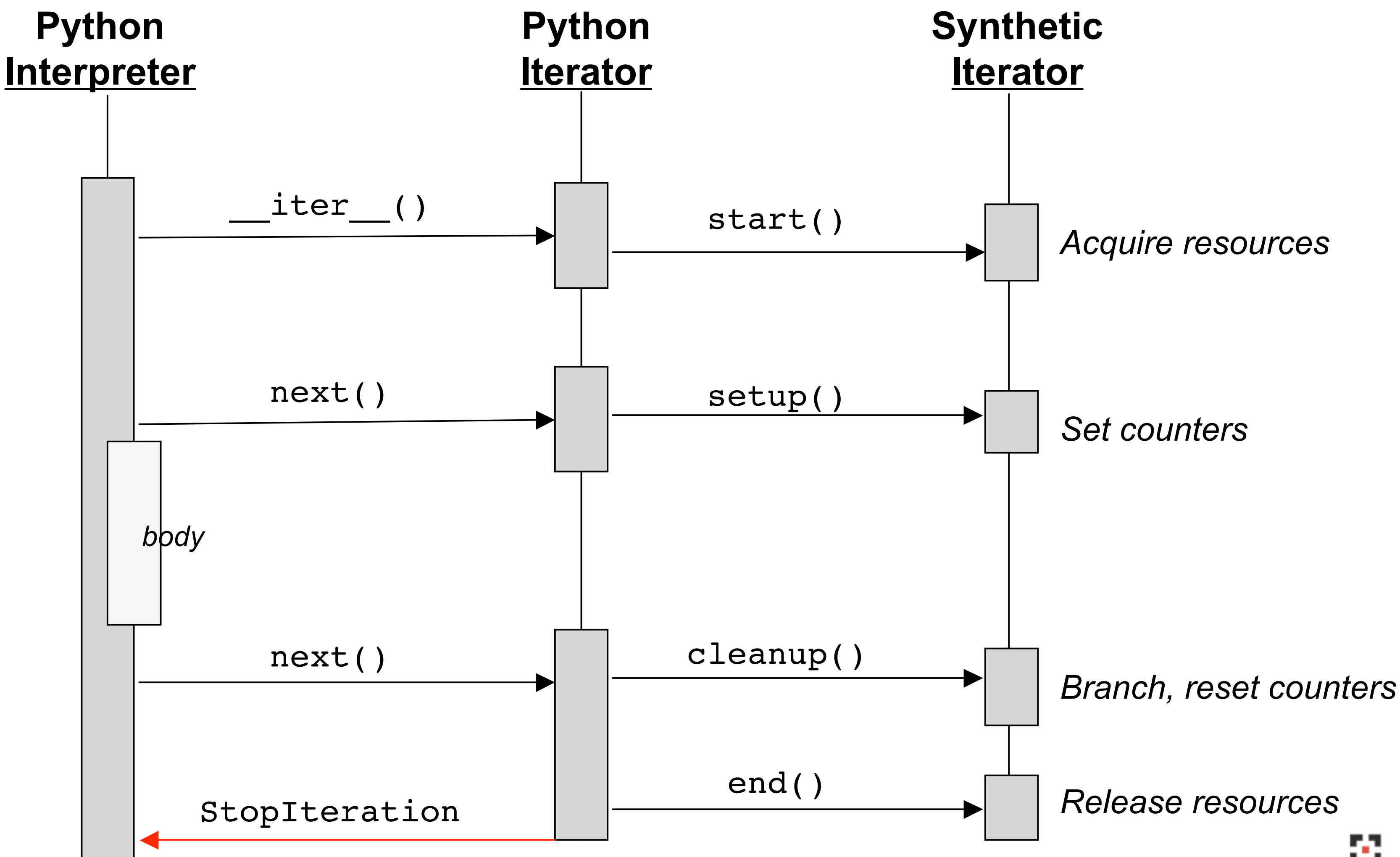
```
# Basic Iteration (PPU/SPU)
a = var(c, 0)

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

proc.execute(c)
--> a = 100
```

How it works…

```
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.

```
a = arange(32)

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

```
# r_* are processor registers, e.g. r_sum = 3
c = InstructionStream()
n = 10000000
a = array(range(n))

# Load a pointer to the array
c.load_word(r_addr, addr(a))

# Set the counter
c.load_word(r_temp, n))
c.add(ppc.mtctr(r_temp))

# Zero the sum
c.add(ppc.addi(r_sum, 0, 0))

# Set a loop label and load the next value
start = c.add(ppc.lwz(r_current, r_addr, 0))


# Update the sum
c.add(ppc.add(r_sum, r_sum, r_current))

# Increment the pointer
c.add(ppc.addi(r_addr, r_addr, 4))

# Decrement the counter and loop
next = code.size() + 1
c.add(ppc.bdnz(-((next - start) * 4)))

result = proc.execute(c)
```

```
# 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.

```
for i in syn_range(c, 10, 20, 2):
  a.v = a + i

--> a = 70
```

```
# X,Y,Z,R are vec_iters

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

--> r[i] = x[i]*z[i] + y[i]
```

High-Performance Iterators

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

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

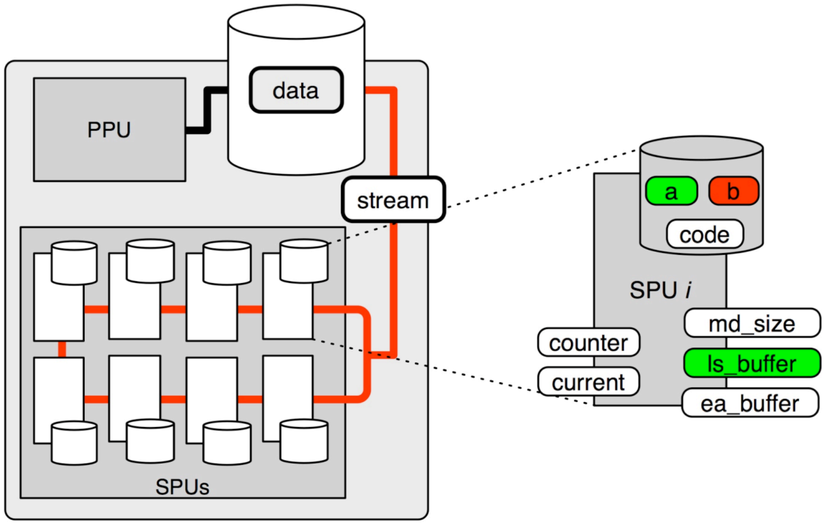
```
# Divide the data for parallel execution
c = ParallelInstructionStream()

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

t1 = proc.execute(c, mode='async', params=[0,2,0])
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.




```
stream = stream_buffer(c, data, buffer_size, buffer_addr,
                              double = True, save = True)
stream = parallel(stream)

for buffer in stream:
  for x in vec_iter(c, buffer):
    x.v = x * x

proc.execute(c, n_spus = 4)
```

Communication

All libspe functions are available to the host program.

```
# 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

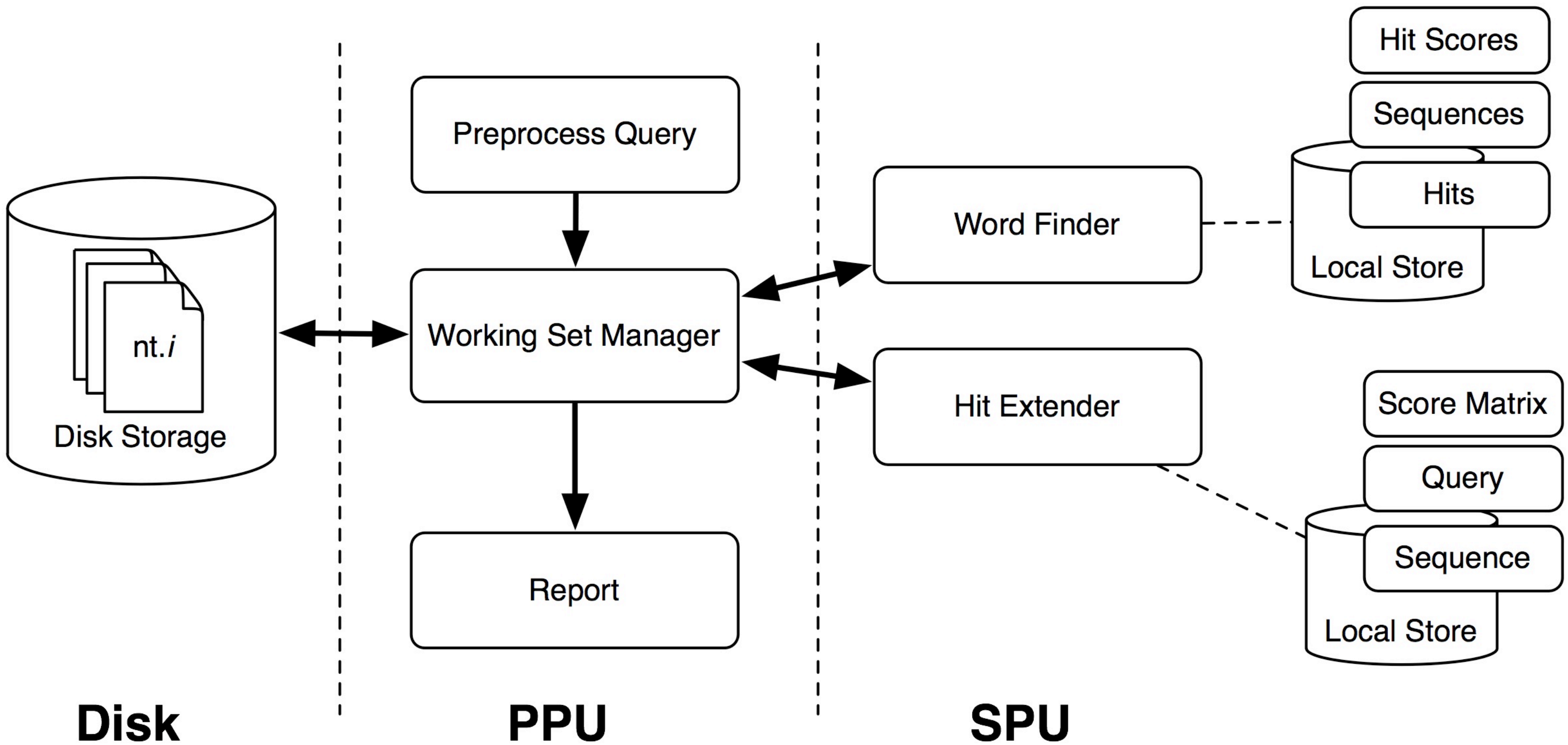
```
AABCDFNNMPMLLASLDA          AAA = 2
                     BCD ×  AAB = -3
                            AAC = 14
                              ...
```

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.

```
           AABCDFNNMPMLLASLDA
ABMNSMMDNSBCDEMLAPPMLL...
           ←       →
```

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

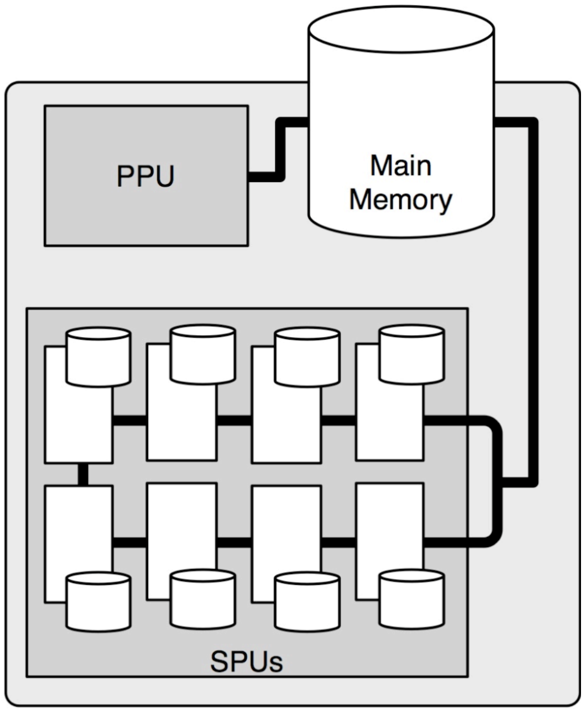
Thank you!

www.synthetic-programming.org

chemuell@cs.indiana.edu

Multi-Core SPE: Cell BE (1)

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



Programmer's view of the Cell BE

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

*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)

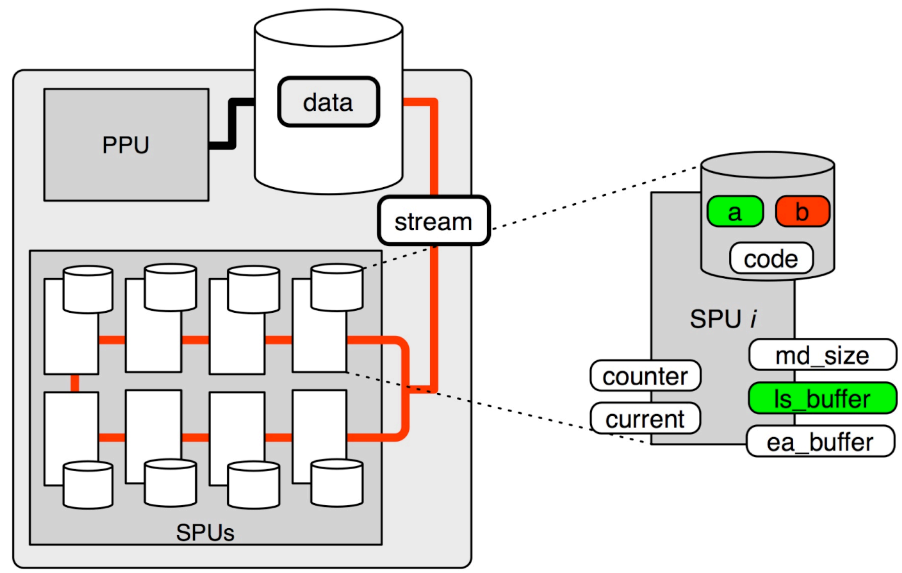
*The Synthetic Programming Environment for the Cell BE includes support for **SPU code synthesis** and **execution**, using both the **SPU ISA** and the higher-level **expression** and **iterator** libraries.*

Double Buffered Stream Operation:

```
c = SPU.InstructionStream()
s = stream_buffer( … )
stream = parallel(s)
md      = memory_desc( … )

for buffer in stream:
  for x in spu_vec_iter(c, md):
    x.v = x + x

r = proc.execute(c, n_spus = 8)
```



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