PyJIT

Dynamic Code Generation From Runtime Data

Simon Burton

Simon.Burton@nicta.com.au

National ICT Australia

Overview

Introduction

PyJIT components

Low Level Virtual Machine (LLVM)

Applications

- Numerical Linear Algebra
- Decision Trees
- Vectorized Operations
- Interval Arithmetic

Conclusion

Introduction

Just-in-time Compilation

- Generate machine code at run-time
- Use "online" knowledge
- Some algorithms then run *faster* than compiled code

Introduction

Pseudo-example

```
for item in stream:
  filter( parameters, item )
```

- apply a filter operation to a stream of data
- filter has parameters set at run-time

Inline the parameters

```
for item in stream:
  filter_1( item )
```

- generate a new function `filter_1`
- compile to machine code

PyJIT Components

$$
\begin{array}{ccccccc}
 & & \textit{Python AST} & & & & \\
 & & \downarrow & & & & \\
 & & \boxed{\text{compiler}} & & & & \\
 & & \downarrow & & & & \\
\textit{low level API} & \rightarrow & \boxed{\text{assembly}} & \xrightarrow[\textit{to ssa}]{} & \boxed{\text{assembly}} & \rightarrow & \textit{LLVM}
\end{array}
$$

Drive PyJIT with python source code

- Operations on native types supported (int, float, etc.)
- Good for numeric processing

Drive PyJIT with low-level constructs

- Construct the *basic blocks*
- Then insert branch and arithmetic instructions

Low Level Virtual Machine

BSD licensed

- Large C++ library
- Can be used as a backend for GCC

Uses simple yet powerful assembly code

- primitive types: integer, floating point
- compound types: structs, arrays, packed (for SIMD)

Optimizes code

- Strength reduction
- Dead code elimination

Generates CPU specific instructions

- SSE, 3dNOW, Altivec
- very fast!

Application 1

Numerical Linear Algebra

- We use the *Portable, Extensible Toolkit for Scientific Computing* (PETSc)
- We construct a PETSc “shell” matrix
- All operations with this matrix are implemented with call-backs

Application 1

Numerical Linear Algebra

- Tri-diagonal Matrix
- Multiplication by this matrix is implemented as a call-back:

```
def mymult(x, y, n):
  y[n-1] = 2.0 * x[n-1] - x[n-2]
  i = n-2
  while i > 0:
    y[i] = 2.0 * x[i] - x[i-1] - x[i+1]
    i = i - 1
  y[0] = 2.0 * x[0] - x[1]
```

Application 1

Numerical Linear Algebra

n	AIJ time	PyJIT time	speedup
$1e4$	480μS	75μS	x6.3
$1e5$	4980μS	373μS	x13
$1e6$	46mS	290μS	x16

- We compare performance with PETSc's sparse matrix
- This is a toy problem

Application 2

Decision Trees: Construction

Input data

- Sequence of training cases
- Each case has a set of attributes, and an “outcome”.

Build a “classifier”

- A tree of `if` statements
- Leaves specify the outcome

Application 2

Decision Trees: example dataset

Sunny	Temp	Humidity	Rain ?
yes	69	70	no
yes	72	95	no
yes	75	70	no
yes	80	90	no
yes	85	85	no
no	64	65	no
no	65	70	yes
no	68	80	yes
no	70	96	yes
no	71	80	yes
no	72	90	no
no	75	80	yes
no	81	75	no
no	83	78	no

Application 2

Decision Trees: example classifier

```
if Sunny == "yes":
    Rain = "no"      # 5 correct
elif Sunny == "no":
    if Temp < 71.5:
        Rain = "yes" # 4 correct, 1 error
    elif Temp >= 71.5:
        Rain = "no"  # 3 correct, 1 error
```

Application 2

Decision Trees: benchmarks

tree size	tree shape	code size	codegen time	C time	PyJIT time	speed up
511	easy	1.5kB	0.35S	18mS	6.7mS	x2.8
511	hard	1.5kB	0.37S	20mS	7.7mS	x2.6
2047	easy	6.0kB	0.80S	27mS	11mS	x2.6
2047	hard	6.0kB	0.81S	36mS	21mS	x1.7
8191	easy	24kB	2.7S	34mS	15mS	x2.3
8191	hard	24kB	2.7S	49mS	29mS	x1.7

- Code generation time is significant
- Use in boosting

Application 3

Vectorized operations: where

`where(a<cutoff,b,c)`

- `a`, `b`, `c` : arrays with the same length
- construct a `result` array with elements from `b` or `c`
- `result[i] = (a[i] < cutoff) ? b[i] : c[i]`

Application 3

Vectorized operations: where

N	Python	NumPy	Psyco	NumExpr	PyJIT
$1e3$	490μS	100μS	45μS	27μS	20μS
$1e4$	5.0mS	780μS	430μS	250μS	120μS
$1e5$	51mS	9.4mS	4.4mS	3.5mS	1.4mS
$1e6$	510mS	94mS	44mS	35mS	13mS

NumExpr is a tiny VM writen in C

- recent work by David M. Cooke, and Tim Hochberg
- for operating on NumPy arrays
- handles simple pointwise calculations
- performs operations blockwise to help caching behaviour

Application 3

Vectorized operations: while-loop

```
def get_weight( cutoff, values, weights, N ):
  i = 0
  weight = 0.0
  while i < N:
    value = values[i]
    if isnan( value )!=0 and value<cutoff:
      weight = weight + weights[i]
    i = i + 1
  return weight
```

- loop over every element of the `values` array
- sum `weights` as we go
- too slow to run this directly in Python

Application 3

Vectorized operations: NumPy

```
def get_weight( cutoff, values, weights ):
    k_mask = -numpy.isnan(values)       # mask of known values
    lt_mask = (values<cutoff) & k_mask
    lt_indices = numpy.nonzero(lt_mask) # array of indices
    lt_weights = weights[ lt_indices ]  # a new array
    return numpy.sum( lt_weights )
```

- Trick: rewrite loop as a succession of *vectorized* operations
- while-loop is now “inside”, at the C-level

Application 3

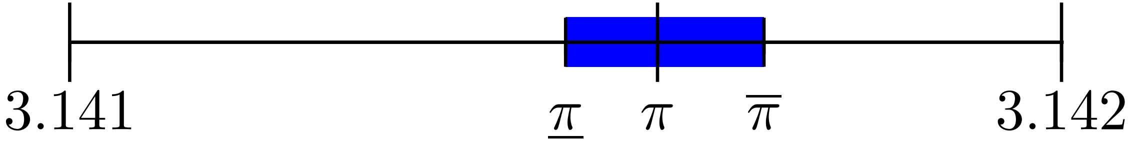
Vectorized operations: benchmarks

N	Python	Psyco	NumPy	PyJIT	speedup
$1e3$	6.6mS	3.5mS	160μS	34μS	x4.7
$1e4$	44mS	31mS	700μS	270μS	x2.6
$1e5$	430mS	300mS	7.2mS	2.8mS	x2.5
$1e6$	4.2S	3.0S	71mS	27mS	x2.7

- Psyco is stumped by the call to `isnan`
- Numpy is more than 50 times faster than the while-loop
- PyJIT applied to the while-loop is faster still

Application 4

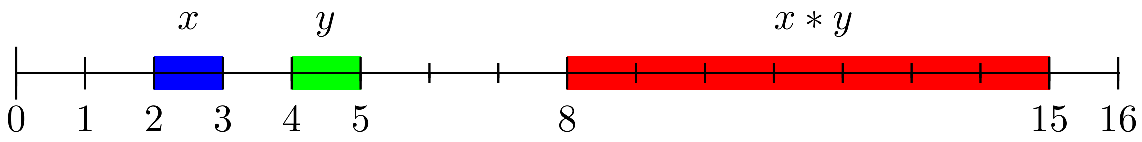
Interval Arithmetic



- Don't just round to the nearest floating point number
- Store a lower and upper bound

Application 4

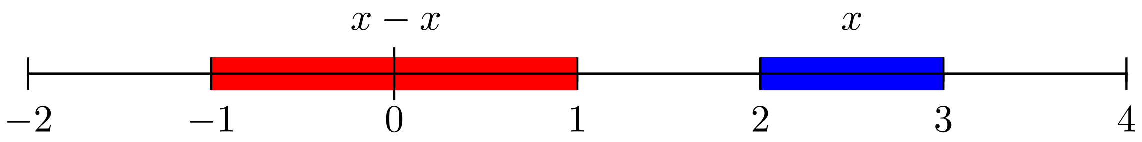
Interval Arithmetic



- carry out operations so that resulting interval encloses all possible values
- Enlarge the result if necessary so that end-points are represented exactly
- Interval calculations amount to a mathematical proof

Application 4

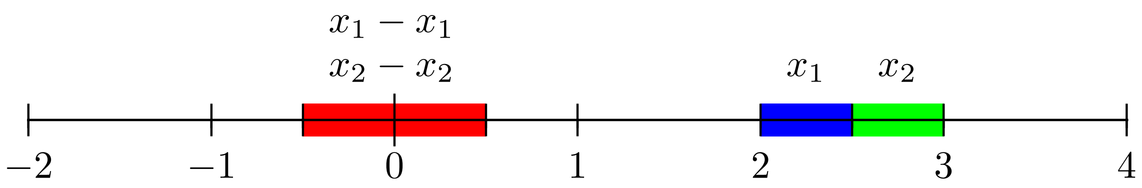
Interval Arithmetic



- Problem: calculations become weak as the intervals explode
- A simple subtraction, $x - x$, is way bigger than it needs to be

Application 4

Interval Arithmetic



- Solution: solve an optimization problem
- Treat calculations *lazily*
- Generate a function on the fly and hand this to an optimization routine
- PyJIT yields similar performance to a compiled version

Conclusion

Data is Code

- Changes the way we think about algorithms
- Inline static data to gain speed increase
- CPU’s are smart: inlining does not always work better

What next ?

- Translate entire python programs (like Psyco)
- Implement NumPy semantics, with optimizations