Vectorization in PyPy’s Tracing Just-In-Time Compiler

Richard Plangger
Andreas Krall

TU Vienna - Institut für Computersprachen

Mai 24, 2016
Outline

Meta-Interpreter
An approach to VM construction

Vectorization Algorithm
High level view to some important details

Embedding it into a TJIT
Details about the implementation

Benchmark Results
Meta-Interpreter
A bird’s eye view

1. Virtual machine for Python
2. Tracing JIT compiler
3. Moving generational GC (Mark and sweep, incremental)
4. Extensible/modular architecture

But we **did not** build a TJIT/GC for Python
JITs are often strongly tied to interpreter & language internals

Components, newly invented over and over

- Bytecode/AST to IR
- Optimizer
- Register allocator
- Code generation

Those are (amongst others) tricky to get right and require a lot of work
RPython

High level language to aid VM construction

1. Import the complete program
   Initialisation can use full Python

2. Process the code object (abstract interpretation)
   Yields control flow and data flow

3. **Annotation**
   Deduce types starting from the interp. entry point

4. **RTyping**
   Converts graphs to low level operations

5. **Codegeneration**
   Emits C code that is later compiled
def LOAD_GLOBAL(self):
    ...

def STORE_FAST(self):
    ...

def BINARY_ADD(self):
    ...

RPYTHON

CODEWRITER

JITCODE

p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
....

p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....

promote_class(p0)
i0 = getfield_gc(p0, 'intval')
promote_class(p1)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(i0, i1)
if (overflowed) goto ...
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
....

compile-time

runtime

ASSEMBLER ← BACKEND ← OPTIMIZER ← META-TRACER
Terminology

- **Translation**: Transforming an interpreter to an executable
- **Tracer**: Attached to the interpreter, records it’s steps
- **Jit Code**: Code the tracer executes to record steps
- **Trace**: Linear sequence of instructions
  Single entry, multi exit
- **Guard**: Instruction to ensure correctness
  Bails out of the trace if it fails
- **Bridge**: A trace that is attached to a guard
  Attaching a bridge is also called “stitching”
Tracing JIT

Trace: List of instructions (Single entry, multi exit)

```python
# Inner Loop
i = 0
while i < X:
    x = func(i * 33)
    if x != 0:
        break
    p[i].x = x // 2
    i += 1
```
Tracing JIT (II)

Procedure of building trace tree is “recursive”

- JitDriver used to be able to trace a dispatch loop
- Entering and leaving another JitDriver supported
- Function tracing
Vectorization Algorithm
Superword parallelism

Element wise addition of two vectors

\[
\begin{array}{c}
42.0 \\
61.8
\end{array}
\begin{array}{c}
+ \\
+
\end{array}
\begin{array}{c}
3.0 \\
38.1
\end{array}
\begin{array}{c}
= \\
=
\end{array}
\begin{array}{c}
45.0 \\
99.9
\end{array}
\]

Single Instruction Multiple Data

\[
\begin{array}{c}
42.0 \\
61.8
\end{array}
\begin{array}{c}
+ \\
+
\end{array}
\begin{array}{c}
3.0 \\
38.1
\end{array}
\begin{array}{c}
= \\
=
\end{array}
\begin{array}{c}
45.0 \\
99.9
\end{array}
\]

Hardware supported (e.g. SSE, NEON, ...)
NumPy, a versatile array processing library

Why does NumPy on PyPy not work out of the box

GC Scheme + C level API of CPython

Solution was needed to optimize the array processing

Potential to use it in regular Python programs
Vectorization

Contradicting goals in a JIT compiler

1. Time & space requirements
2. Traces instead of regions
3. Specifically targeting SIMD instructions

Traces might contain enough parallelism
Building blocks

1. Loop unrolling
   if necessary
2. Data structures to express dependencies
3. Analysis step
4. Transformation
5. Adapted code generation
Data Dependency

Graph representation of instruction dependencies.

- True dependency
- Anti dependency
- Output dependency
- (Control dependency)
Ubiquitous guard exists

Introduces many dependencies
Need to be eliminated, no validity preserving transformation possible

Code motion moves guard as early as possible
Move a path of pure operations earlier
Instruction Parallelism

Analysis step to group instructions
A simple greedy comparison of operations initiated by load/store operations

- $O(n^2)$ to compare each instruction with another
  
  $n$ ... # trace instructions

- Only load/store instructions are considered at first

- Extension phase follows dependencies
  
  To reveal other parallel instructions

Grouping of “isomorphiс” instructions
Same IR opcode and argument types
Instruction Parallelism (II)

Resulting information contains

1. **Pairs**: Tuple of parallel isomorphic instructions
2. **Pack**: N-Tuple of parallel isomorphic instructions

Transformation pass
Can be done by re-scheduling the trace considering pairs and packs

Code generation
Scheduling

Work through the dependency graph:

1. Pick, remove and emit a schedulable node
2. Remove edges and recompute the set of schedulable nodes

Pack dependencies
Cycle can only be broken by partly/fully removing the pack restriction
Additional enhancements done while scheduling:

1. Vector cropping. Size of the input vector is too big/small
   - Integer sign extensions

2. Vector slot movement
   - Conversion 32-bit float to 64-bit float

3. Invariant scalar/constant expansion

4. Inline scalar/constant expansion
Example

1 \( i = 0 \)
2 \textbf{while} \( i < R \):
3 \hspace{1em} b[i] = a[i] + 1
4 \hspace{1em} i = i + 1
5

1 \( x = \text{load}(a, i) \)
2 \( z = x + 1 \)
3 \( \text{store}(b, i, z) \)
4 \( \text{guard}(i+1 < R) \)
5 \# iteration \( i+1 \)
6 \( y = \text{load}(a, i+1) \)
7 \( w = y + 1 \)
8 \( \text{store}(b, i+1, w) \)
9 \( \text{guard}(i+2 < R) \)
10
Example

```
1 i = 0
2 while i < R:
3     b[i] = a[i] + 1
4     i = i + 1
```

```
1 x = load(a,i)
2 z = x + 1
3 store(b,i,z)
4 guard(i+1 < R)
5 # iteration i+1
6 y = load(a,i+1)
7 w = y + 1
8 store(b,i+1,w)
9 guard(i+2 < R)
```

Iteration i

- \( x = \text{load}(a, i) \)
- \( z = x + 1 \)
- \( \text{store}(b, i, z) \)
- \( \text{guard}(i < R) \)

Iteration i+1

- \( y = \text{load}(a, i+1) \)
- \( w = y + 1 \)
- \( \text{store}(b, i+1, w) \)
- \( \text{guard}(i+1 < R) \)
Tracing complications

1. \( x = \text{load}(a, i) \)
2. \( z = x + 1 \)
3. \( \text{store}(b, i, z) \)
4. \( \text{guard}(i+1 < R) \)
5. \( y = \text{load}(a, i+1) \)
6. \( w = y + 1 \)
7. \( \text{store}(b, i+1, w) \)
8. \( \text{guard}(i+2 < R) \)

? Store operations independent

Counter example: Guard fails, but \( \text{store}(b, i+1, w) \) already executed.

☀ Valid to execute guard earlier
“Guard Early Exit”

1. $\text{guard}(i + 1 < R)$
2. $\text{guard}(i + 2 < R)$
3. $x = \text{load}(a, i)$
4. $z = x + 1$
5. $\text{store}(b, i, z)$
6. $\# \text{guard}(i + 1 < R)$
7. $y = \text{load}(a, i + 1)$
8. $w = y + 1$
9. $\text{store}(b, i + 1, w)$
10. $\# \text{guard}(i + 2 < R)$

Move guards to an earlier place. Scheduling reorders instructions.
“Guard Early Exit”

Move guards to an earlier place. Scheduling reorders instructions.

Pure operations must precede guards
guard( i+1 < R)
guard( i+2 < R)
guard( i+3 < R)
x = load(a, i)
z = x + 1
store(b, i, z)
y = load(a, i+1)
w = y + 1
store(b, i+1, w)
v = load(a, i+2)
q = v + 1
store(b, i+2, q)
1  guard(i+1 < R)
2  guard(i+2 < R)
3  guard(i+3 < R)
4  x  =  load(a, i)
5  z  =  x  +  1
6  store(b, i, z)
7  y  =  load(a, i+1)
8  w  =  y  +  1
9  store(b, i+1, w)
10 v  =  load(a, i+2)
11 q  =  v  +  1
12 store(b, i+2, q)
guard(i+1 < R)
guard(i+2 < R)
guard(i+3 < R)
x = load(a, i)
z = x + 1
store(b, i, z)
y = load(a, i+1)
w = y + 1
store(b, i+1, w)
v = load(a, i+2)
q = v + 1
store(b, i+2, q)
Packing

1. guard(i+1 < R)
2. guard(i+2 < R)
3. guard(i+3 < R)
4. x = load(a, i)
5. z = x + 1
6. store(b, i, z)
7. y = load(a, i+1)
8. w = y + 1
9. store(b, i+1, w)
10. v = load(a, i+2)
11. q = v + 1
12. store(b, i+2, q)

Packs are a representation of vector instructions

1. Independent instructions
2. Isomorphic instruction pairs/packs
Vector loop

1. label(a,b,i,R)
2. guard(i+3 < R)
3. \[x,y,v] = \text{vec_load}(a,i)\]
4. \[z,w,q] = [x,y,v] + [1,1,1]\]
5. \text{vec_store}(b,i,[z,w,q])
6. jump(a,b,i+3,R)
Embedding it into PyPy
Embedding it into PyPy

Optimization just before backend assembly

• Just after the “unrolling optimization”
  Guard strength reduction, invariant code motion, object virtualization

Roughly 4000 lines of code
+ 4000 for testing
Accumulation

Reduction cannot be represented
Need to carry information out of trace loops and recognize the pattern

*Chained computations can be matched, saved as accumulation pack*

1. Use an accumulation vector to save the computation
   In each slot only a part of the information is stored

2. Several points need the resulting value
   “Flush” the real value (e.g. sum: horizontal add)

3. Scheduling pass needs to be adapted slightly
Speculative ABC optimization

Array bound checks are not fully eliminated
Loop bounds and array bounds are checked

Speculative step to remove guarding instructions.

If the loop bound is smaller than the length of the array, no IndexError cannot occur on that array.
Transitive relation introduced that is checked before the vector loop
Version trace loops

Switch back to interpreter always necessary?

Several iterations needed to complete the loop (odd vector length)

1 Directly attach versions of the loop to the loop exit
2 As well as to guards for ABC

No need to switch back to the interpreter
Extensions

The following has been added:

- Constant/Scalar expansion
- Accumulation
- Speculative ABC optimization for array accesses
- Trace Loop versioning

Future work

- Aligned memory access not yet supported
- No reordering support of interleaved formats
Evaluation
### Optimization time

<table>
<thead>
<tr>
<th>Count</th>
<th>Instruction count</th>
<th>Unroll factor</th>
<th>Microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12-16</td>
<td>2</td>
<td>101.47</td>
</tr>
<tr>
<td>5</td>
<td>17-19</td>
<td>4</td>
<td>158.46</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>8</td>
<td>224.03</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>16</td>
<td>396.60</td>
</tr>
</tbody>
</table>
## Benchmark programs

<table>
<thead>
<tr>
<th>Name</th>
<th>CPython</th>
<th>PyPy</th>
<th>VecOpt</th>
<th>VecOpt Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc-distance</td>
<td>0.07898</td>
<td>0.1813</td>
<td>0.1608</td>
<td>1.1</td>
</tr>
<tr>
<td>diffusion</td>
<td>0.5603</td>
<td>5.665</td>
<td>3.889</td>
<td>1.5</td>
</tr>
<tr>
<td>evolve</td>
<td>0.1967</td>
<td>1.815</td>
<td>1.728</td>
<td>1.1</td>
</tr>
<tr>
<td>fft</td>
<td>0.9507</td>
<td>0.2981</td>
<td>0.2955</td>
<td>1.0</td>
</tr>
<tr>
<td>harris</td>
<td>0.3485</td>
<td>3.119</td>
<td>1.504</td>
<td>2.1</td>
</tr>
<tr>
<td>l2norm</td>
<td>0.564</td>
<td>1.73</td>
<td>1.634</td>
<td>1.1</td>
</tr>
<tr>
<td>lstsqr</td>
<td>0.3844</td>
<td>1.506</td>
<td>1.39</td>
<td>1.1</td>
</tr>
<tr>
<td>multiple-sum</td>
<td>0.1432</td>
<td>0.6341</td>
<td>0.25</td>
<td>1.1</td>
</tr>
<tr>
<td>rosen</td>
<td>0.5795</td>
<td>3.498</td>
<td>3.438</td>
<td>1.0</td>
</tr>
<tr>
<td>specialconvolve</td>
<td>0.4713</td>
<td>3.876</td>
<td>2.649</td>
<td>1.5</td>
</tr>
<tr>
<td>vibr-energy</td>
<td>0.2784</td>
<td>0.7552</td>
<td>0.699</td>
<td>1.1</td>
</tr>
<tr>
<td>wave</td>
<td>2.191</td>
<td>1.114</td>
<td>1.166</td>
<td>0.9</td>
</tr>
<tr>
<td>wdist</td>
<td>2.927</td>
<td>1.202</td>
<td>1.179</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Python programs

Andreas Krall, Richard Plangger
Questions?