py2llvm: Python to LLVM translator

Syoyo Fujita

Agenda

- Motivation
- How it works
- Performance
- Limitation
- Conclusion

Agenda

- Motivation
- How it works
- Performance
- Limitation
- Conclusion

py2llvm

- **Python シンタックスを LLVM に変換**
  - Python インタプリタでも動くし,
  - コンパイルして高速に実行することもできる

Prototyping phase

デバッグ、
アルゴリズム開発
は python で

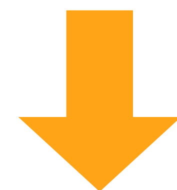
python

python interpreter

Production phase

LLVM codegen

リリースコードは
高速実行



native code

Motivation 1/4

C で数値演算、グラフィックスなどのパフォーマンス指向コードを書くのはめんどくさい

アルゴリズム変更時の書き直しの手間がおおきい.

とくに SIMD 命令を扱う場合

アラインの問題、命令セットの問題, ...

とある SIMD コードからの抜粋

```
    /* calculate u, v and t for all triangles. */
    const __m128 uu = _mm_mul_ps(dot_sse(sx, sy, sz, px, py, pz), rpa);
    const __m128 vv = _mm_mul_ps(dot_sse(rdx, rdy, rdz, qx, qy, qz),
rpa);

    __m128 result;

    a = _mm_and_ps(
        _mm_and_ps(
#if 0   /* original code. no buck face culling */
            _mm_cmpgt_ps(_mm_mul_ps(a, a), eps2),
#else   /* do back face culling */
            _mm_cmpgt_ps(a, eps),
#endif
            _mm_cmpngt_ps(_mm_add_ps(uu, vv), one)
        ),
        _mm_and_ps(
            _mm_cmpnlt_ps(uu, zero),
            _mm_cmpnlt_ps(vv, zero)
        )
    );
```

解説なしで
誰が理解で
きようか

いやできまい

(反語)

こういう感じでSIMDコーディングできればなぁ...

```
def add_func():

    a = vec([1.0, 2.0, 3.0, 4.0])
    b = vec([0.1, 0.2, 0.3, 0.4])

    return a + b
```

Motivation 2/4

Python

- プログラムが簡潔に書ける
- でも実行速度はとてつもなく遅い
- psyco, pypy -> 思うほど早くならない
- ShedSkin -> 最適! というほどではない

Motivation 3/4

Python のように簡潔に書けるけど、実行は
ちょー早いようにできないものか

Motivation 4/4

Python を DSL のように扱い、LLVM コードを吐くようにしてはどうか

まず Python インタプリタで実行してラピッドプロトタイピング.

完成したら LLVM に変換して高速実行.

Agenda

- Motivation
- How it works
- Performance
- Limitation
- Conclusion

やってみた.

SIMD と fp 演算の変換を重視した

使うライブラリ

- **Python compiler モジュール**
  - Python コードをパースしてAST を作ってくれる. Python 標準ライブラリ
- **llvm-py**
  - LLVM API への Python ラッパー
- **LLVM**

compiler module 1/2

```
from MUDA import *

def add_func(a = vec, b = vec):
    return a + b
```

```
import compiler

ast = compiler.parseFile(sys.argv[1])
print ast
```

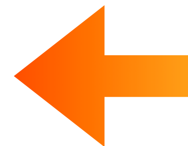
```
Module(None, Stmt([From('MUDA', [('*', None)], 0),
Function(None, 'add_func', ['a', 'b'], [Name('vec'),
Name('vec')], 0, None, Stmt([Return(Add((Name('a'),
Name('b'))))]))]))
```

compiler module 2/2

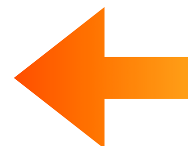
Visitor パターンで AST をトラバースしてくれる.

```
compiler.walk(ast, CodeGenLLVM())
```

```
class CodeGenLLVM():
  def visitModule(self, node):
    ...
  def visitFunction(self, node):
    ...
  ...
```



Module ノードをトラバースしたときに実行するメソッドを記述



Function ノードをトラバースしたときに実行するメソッドを記述

llvm-py

```
from llvm.core import *

module = Module.new("my_module")

ty_double = Type.double()
ty_int    = Type.int()
ty_func   = Type.function( ty_int, [ ty_double, ty_double ] )

func      = Function.new( module, ty_func, "foobar" )

func.args[0].name = "arg1"
func.args[1].name = "arg2"

entry = func.append_basic_block("entry")

builder = Builder.new(entry)

tmp1 = builder.add(func.args[0], func.args[1], "tmp1")
```

How translation works

```
def func():
  return a + b;
```

python

compiler.parseFile()

```
Function(Return(A
dd(Name(a),
Name(b)))
```

python ast

型推論

シンボル解決

```
a: int
b: int
```

python ast

llvm-py で

codegen

```
declare @func() {
  %tmp = add %a, %b
  ret %tmp
}
```

LLVM IR

モジュール構成

SymbolTable.py	シンボル管理
TypeInference.py	型推論
MUDA.py	ベクトル型定義
CodeGenLLVM.py	llvm コード生成

型推論

Python は動的言語なので型は明確に定義しなくていい

どう変数の型を(静的に)解決するか?

型推論で判定

```
a = 1
b = 1.0
c = vec([1.0, 2.0, 3.0, 4.0])
```

a : int だと分かる

b : float だと分かる

c : ベクトル型(MUDA.py で定義)だと分かる

```
a = 1
b = 3
c = a + b
```

a : int だと分かる

b : int だと分かる

c : ？

「a が int で b が int なら
c も int じゃね」と心の
中で思ったならッ！！
そのときステに c の型は
決まっているんだッ！

無理なケース

```
def add_func(a, b):
  return a + b

add_func(3,2)
```

a, b : 型を決められない.

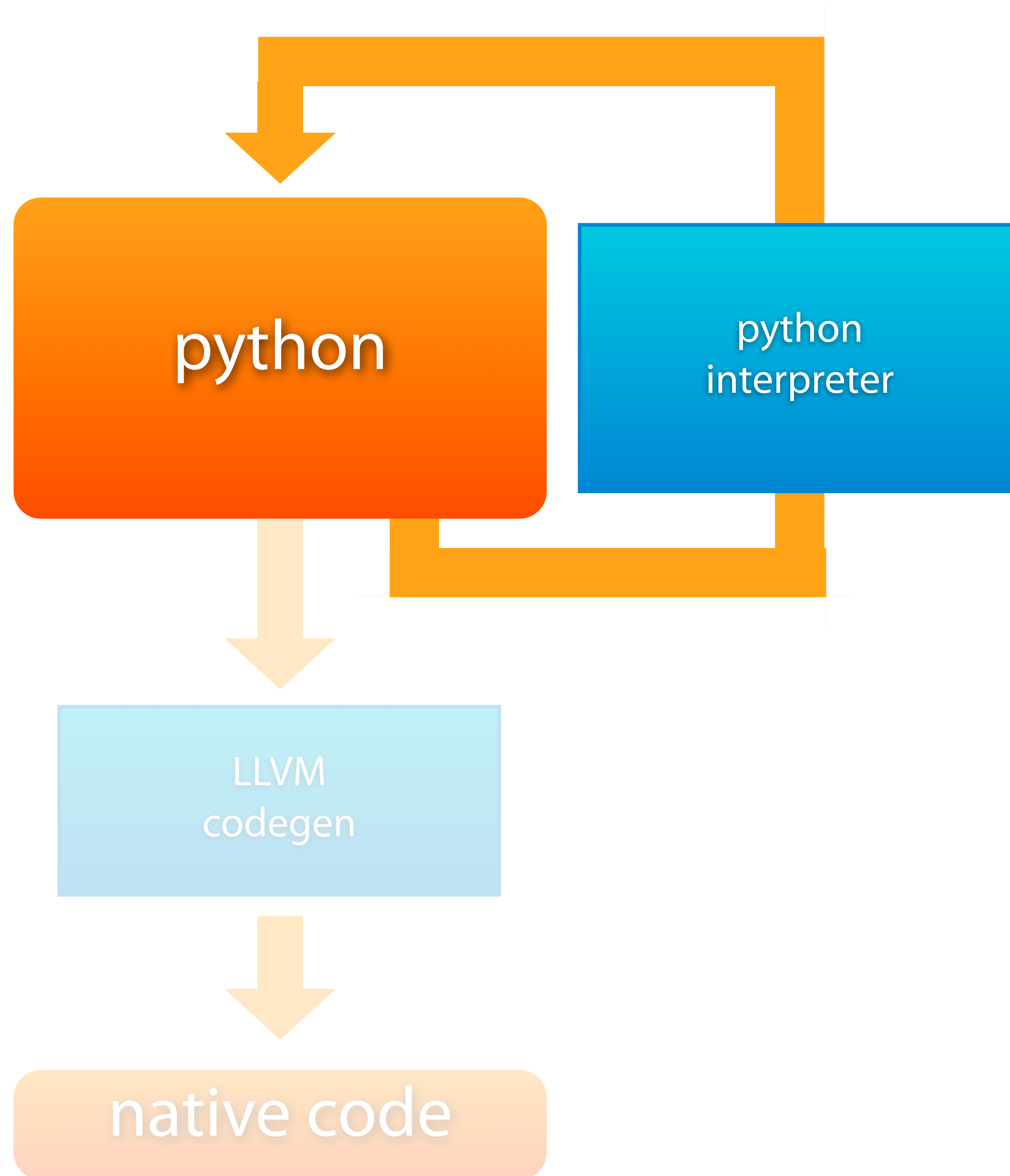
(関数のバウンダリを越えての型推論は未実装)

```
def add_func(a = int, b = int):
    return a + b
```

引数の型はデフォルト値で与えるようにする、という制約を付けるようにした

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

がどう処理されるか見ていく

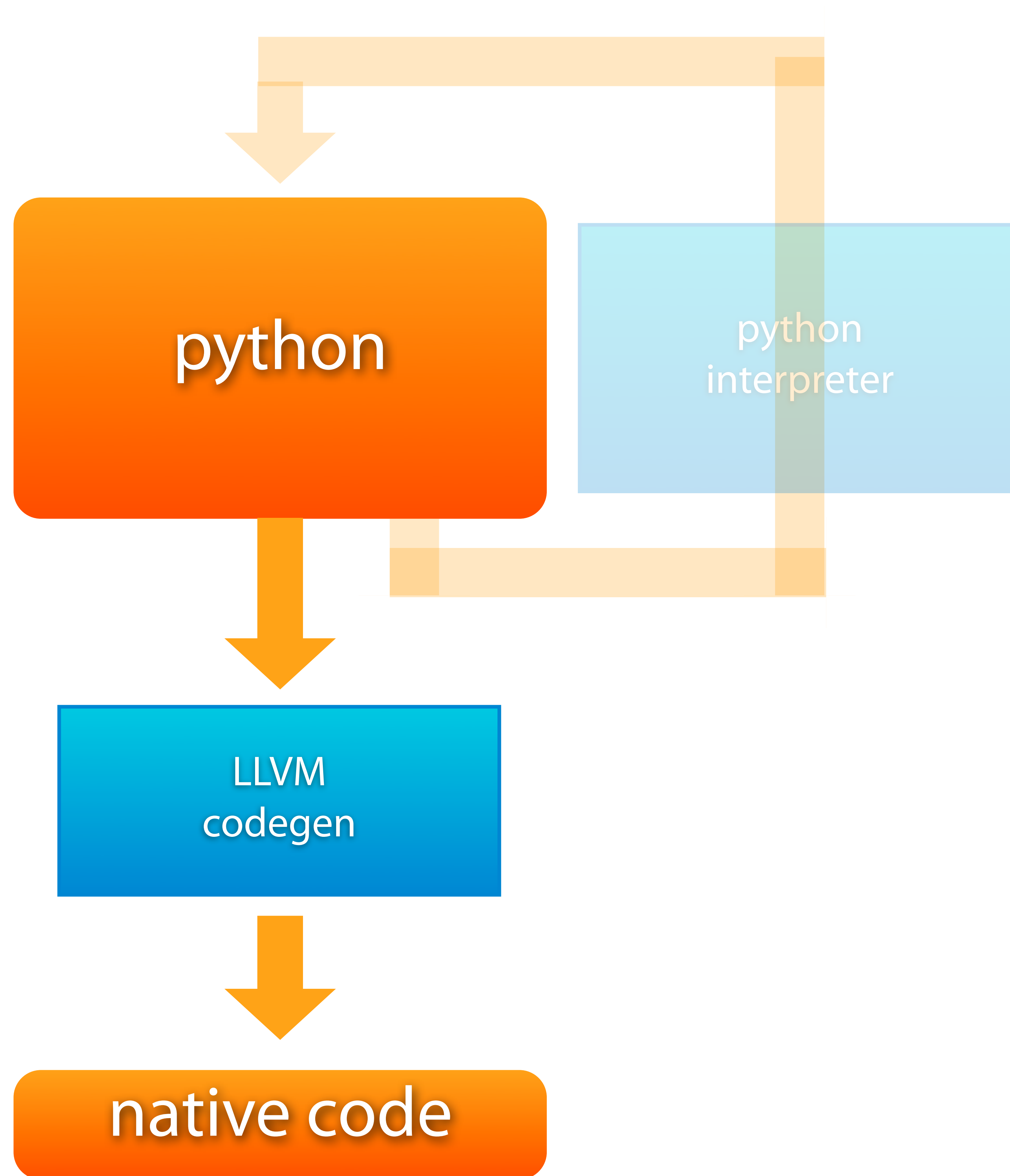

python
python
interpreter
LLVM
codegen
native code

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

vec 型は MUDA モジュールで定義されているクラス

演算はオーバーロードで対応

```
class vec(object):
   value = []
   def __add__(self, b):

        tmp = vec([x + y for x, y in zip(self.value,
b.value)])

        return tmp
```


python
python
interpreter
LLVM
codegen
native code

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

compiler.parseFile() で AST を作成

```
Function(None,
         'add_func',
        ['a', 'b'],
        [Name('vec'), Name('vec')], 0, None,
   Stmt([Assign([AssName('c', 'OP_ASSIGN')],
          Add((Name('a'), Name('b')))),
         Return(Name('c'))]))
```

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

引数 a, b は vec 型だと分かる.
a, b をシンボルテーブルに登録.
引数をレジスタにロードするコードを出力.

```
%tmp5 = alloca <4 x float>
store <4 x float> %a, <4 x float>* %tmp5
%tmp6 = alloca <4 x float>
store <4 x float> %b, <4 x float>* %tmp6
%tmp7 = load <4 x float>* %tmp5
%tmp8 = load <4 x float>* %tmp6
```

```
Function(None,
         'add_func',
         ['a', 'b'],
         [Name('vec'), Name('vec')], 0, None,
    Stmt([Assign([AssName('c', 'OP_ASSIGN')],
              Add((Name('a'), Name('b')))),
         Return(Name('c'))]))
```

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

a と b の加算. a と b の型は？
-> 型推論で a : vec, b : vec だと分かる(シンボルテーブルを引く)

```
%tmp9 = add &lt;4 x float&gt; %tmp7, %tmp8
```

```
Function(None,
         'add_func',
        ['a', 'b'],
        [Name('vec'), Name('vec')], 0, None,
    Stmt([Assign([AssName('c', 'OP_ASSIGN')],
           Add((Name('a'), Name('b')))),
         Return(Name('c'))]))
```

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

c への代入. c の型は？
右辺が vec なので左辺の c も vec だと分かる

```
%c = alloca <4 x float>
store <4 x float> %tmp9, <4 x float>* %c
```

```
Function(None,
         'add_func',
        ['a', 'b'],
        [Name('vec'), Name('vec')], 0, None,
    Stmt([Assign([AssName('c', 'OP_ASSIGN')],
           Add((Name('a'), Name('b')))),
         Return(Name('c'))]))
```

```
from MUDA import *

def add_func(a = vec, b = vec):
    c = a + b
    return c
```

c は vec 型なので関数の戻り値の型も vec だと分かる
(ここでやっと関数の戻り値の型が分かる)

```
%tmp10 = load <4 x float>* %c
ret <4 x float> %tmp10
```

```
Function(None,
         'add_func',
        ['a', 'b'],
        [Name('vec'), Name('vec')], 0, None,
    Stmt([Assign([AssName('c', 'OP_ASSIGN')],
           Add((Name('a'), Name('b')))),
         Return(Name('c'))]))
```

関数の戻り値の型

return の戻り値の型から決定

return 式が見つかるまでわからない

いったん関数をパースし、戻り値の型を求め、再度関数をパースしている

2 pass の処理

出力されたLLVMコード

```
$ cat add.ll

define <4 x float> @add_func(<4 x float> %a, <4 x float> %b) {
entry:
    %tmp5 = alloca <4 x float>        ; <<4 x float>*> [#uses=2]
    store <4 x float> %a, <4 x float>* %tmp5
    %tmp6 = alloca <4 x float>        ; <<4 x float>*> [#uses=2]
    store <4 x float> %b, <4 x float>* %tmp6
    %tmp7 = load <4 x float>* %tmp5       ; <<4 x float>> [#uses=1]
    %tmp8 = load <4 x float>* %tmp6       ; <<4 x float>> [#uses=1]
    %tmp9 = add <4 x float> %tmp7, %tmp8     ; <<4 x float>> [#uses=1]
    %c = alloca <4 x float>       ; <<4 x float>*> [#uses=2]
    store <4 x float> %tmp9, <4 x float>* %c
    %tmp10 = load <4 x float>* %c     ; <<4 x float>> [#uses=1]
    ret <4 x float> %tmp10
}
```

最適化してみる

```
$ llvm-as < add.ll | opt -std-compile-opts -f | llvm-dis

; ModuleID = '<stdin>'

define <4 x float> @add_func(<4 x float> %a, <4 x float> %b) nounwind
{
entry:
    %tmp9 = add <4 x float> %a, %b       ; <<4 x float>> [#uses=1]
    ret <4 x float> %tmp9
}
```

ネイティブコードにしてみる

```
$ llvm-as < add.ll | opt -std-compile-opts -f | llc

    .text
    .align  4,0x90
    .globl  _add_func
_add_func:
    addps   %xmm1, %xmm0
    ret

    .subsections_via_symbols
```

まさに、、、

最ッ!…適ッ!…

Agenda

Motivation

How it works

Performance

Limitation

Conclusion

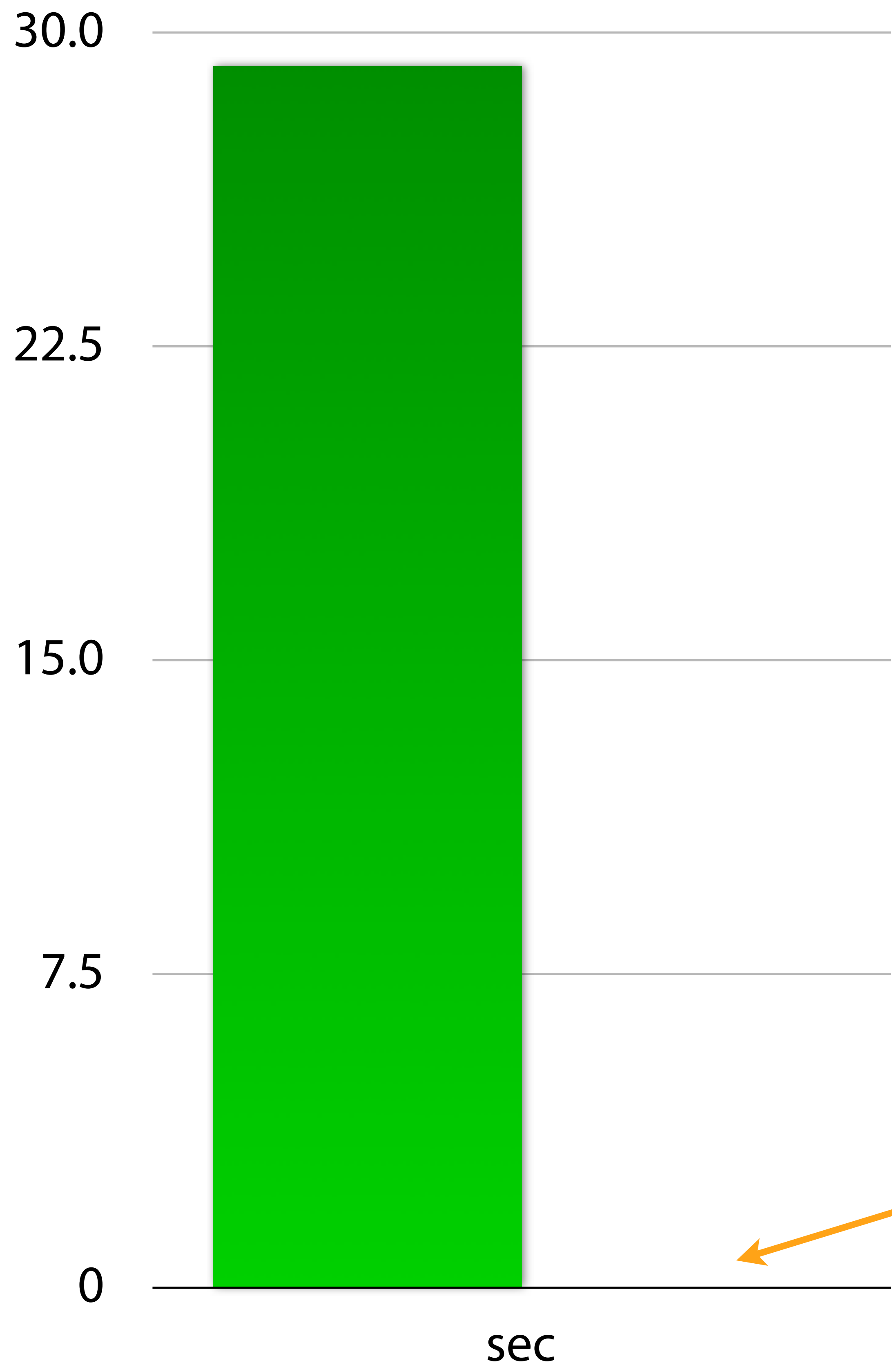
パフォーマンス測定

```
def BlackScholes(S = vec, X = vec, T = vec, R = vec, V = vec):

    sqrtT  = vsqrt(S)

    d1     = (vlog(S / X) + (R + vec(0.5) * V * V) *T) / (V * sqrtT)
    d2     = d1 - V * sqrtT

    cnd_d1 = cnd(d1)
    cnd_d2 = cnd(d2)

    expRT  = vexp(vec(-1.0) * R * T)

    retCall = S * cnd_d1 - X * expRT * cnd_d2

    return retCall
```

BlackSholes を 5 万回実行したときの時間

実行マシン: Intel Mac Core2 2.16 GHz

python llvm



Faster

python	29.2 sec
python -> llvm -> native	0.03 sec

可視化できない....

1,000 倍の
高速化

それはひょっとし
てギャグで言って
いるのか?

Investigation

llvm で吐かれた

BlackScholes

関数は

最適化されて

350 命令に

```
subl    $540, %esp
movaps  %xmm0, 432(%esp)
movaps  %xmm1, 416(%esp)
movaps  %xmm2, 400(%esp)
movaps  %xmm3, 384(%esp)
movss   %xmm0, (%esp)
call    L_sqrtf$stub
fstpt   372(%esp)
pshufd  $1, 432(%esp), %xmm0
movss   %xmm0, (%esp)
call    L_sqrtf$stub
fstpt   360(%esp)
movaps  432(%esp), %xmm0
movhlps %xmm0, %xmm0
movss   %xmm0, (%esp)
call    L_sqrtf$stub
fstpt   348(%esp)
pshufd  $3, 432(%esp), %xmm0
movss   %xmm0, (%esp)
call    L_sqrtf$stub
fstpt   336(%esp)
movaps  432(%esp), %xmm0
divps   416(%esp), %xmm0
...
```

理論値

350 [Insts] * 50 [kloop] = 17.5 [Mcycle]

17.5 [Mcycle] / 2.16 [GHz] = 0.08 sec

*) add, mul 同時実行はひとまず考えない

*) 最新 Core2 は基礎的な SIMD 演算命令を
1 cycle で実行できる.

理論値との乖離

0.03(実測) / 0.08(理論値) = 3.75

4 倍ほど理論値より離れている

理論値を破ってしまうほどおかしな値ではない.

実測値から計算すると 1300 cycles/関数

Instruction 内訳

4倍遅い理由

いくつかの数学関数のコールが各10 ~ 100サイクルかかるのでそれが影響している

Inst	**num**
sqrtf	4
logf	4
expf	12
divps	4
Other	326
Total	**350**

SIMD 数学関数

完全 SIMD 化された数学関数を使うようにすればさらなる高速化は可能

MUDA に sqrt, exp, log の SIMD 版が実装されている http://lucille.atso-net.jp/blog/?p=497

こちらを使うと 1 BlackScholes 関数が 1,000 サイクル(flops). 5 万回繰り返し時 = 0.023 sec

Swizzle

- ベクトル要素の取り出し、並べ替えを行う
- expression
- コーディングが非常に楽になる

```
a = vec([1.0, 2.0, 3.0, 4.0])

a.x       # => 1.0
a.wzyx    # => [4.0, 3.0, 2.0, 1.0]
a.yyyy    # => [2.0, 2.0, 2.0, 2.0]
```

Python でやるには?

```
class vec():

    ...

    def setx(self): ...
    sef getx(self): ...
    x = property(setx, getx)

    def sety(self): ...
    sef gety(self): ...
    y = property(sety, gety)

    ...
```

4^1 + 4^2 + 4^3 + 4^4 = 340 個も

書かなければならない!

__getattr__ を使う

```
class vec():

    ...

    def __getattr__(self, name):

        d = { 'x' : 0, 'y' : 1, 'z' : 2, 'w' : 3 }

        assert len(name) < 5, "Invalid attribute: %s" % name

        if len(name) == 1:
            return self.value[d[name]]

        v = vec([0.0, 0.0, 0.0, 0.0])

        for (i, s) in enumerate(name):
            if not d.has_key(s):
                raise Exception("Invalid letter for swizzle:", name)

            v.value[i] = self.value[d[s]]

        for i in range(len(name), 4):
            v.value[i] = self.value[d[name[-1]]]

        return v
```

Agenda

Motivation

How it works

Performance

Limitation

Conclusion

制限 1/2

- **すべての Python 機能が使えるわけではない**
  - 動的言語の性質のたぐいは使えない(実行しないと型が分からないのはダメ)
  - OO の機能もなし

制限 2/2

演算精度

- python の float は内部では double
- py2llvm は float(fp32) にしている.
  - 演算結果が必ずしも正確に一致しない.

Agenda

Motivation

How it works

Performance

Limitation

Conclusion

まとめ

Python シンタックスから LLVM コードへ変換

Python でもそのまま動くし、変換してネイティブ実行もできる

ネイティブ変換効率はとても高い. C 最適実装と同じくらいの速度

静的に解決できるコードのみ変換可能

SIMD fp 演算式の変換を重視

http://code.google.com/p/py2llvm/

Future work 1/2

配列のサポート

構造体のサポート

さらなる数学関数や外部関数呼び出しのサポート

Future work 2/2

- **py2llvm という名前がよくない**
  - Python コードがすべて変換できるわけではないし.
  - optimized python -> oppy ?...
  - Psyga? -> 日本最大のコングロリマット企業の名前とかぶる...

?