LLVM Optimized Python for Scientific Computing

Harvard-Smithsonian Center for Astrophysics

Stephen Diehl (@smdiehl)
- NumPy
- SciPy
- Anaconda
- Numba
- AstroPy
- Scikit-Learn
- Skimage
- Cython
- Theano
- Julia
- Fortran
SciPython = An Embedded DSL for Composing Kernels
SciPython = Shell Scripting + C + Fortran
Python is unreasonably effective.

- Overloaded operators.
- A well documented C API.
- Buffer protocol.
- Community leaders.
Python is unreasonable.

- Python is a language stuck in 1973 forever.
- Many people in the Python community don’t understand the needs of numerical programmers.
- Can’t expect the language to evolve, but the utility of the tools we have are world-class.
Instead of writing your numeric kernels in C / Fortran / Cython you write idiomatic Python, transform it using LLVM toolchain to custom generate efficient machine code which can be invoked from the Python runtime.

The goal shared among some people is to migrate a lot of the growing pile of legacy Cython in projects skimage and scikit-learn and lift this more sustainable idiomatic Python. Let domain experts (astrophysicists) focus on the problem domain, not low-level optimizations or cross-language integration.

- More Python, less C++.
- FYI: The vision may take a while.
- http://numfocus.org/
LLVM is a statically typed intermediate representation and an associated toolchain for manipulating, optimizing and converting this intermediate form into native code.

- Julia
- Clang
- Rust
- Numba
- Haskell
- Cuda

Adobe, AMD, Apple, ARM, Google, IBM, Intel, Mozilla, Nvidia, Qualcomm, Samsung, Xilinx
Types map directly to optimal machine implementation, platform agnostic!

```
i1 1 ; boolean bit
i32 299792458 ; integer
float 7.29735257e-3 ; single precision
double 6.62606957e-34 ; double precision

[10 x float] ; Array of 10 floats
[10 x [20 x i32]] ; Array of 10 arrays of 20 integers.

<8 x double> ; Vector of 8 double
define i32 @test1(i32 %x, i32 %y, i32 %z) {
  %a = and i32 %z, %x
  %b = and i32 %z, %y
  %c = xor i32 %a, %b
  ret i32 %c
}
test1:

.cfi_startproc
andl %edx, %esi
andl %edx, %edi
xorl %esi, %edi
movl %edi, %eax
ret
int count(int n)
{
    int i = 0;
    while(i < n)
    {
        i++;
    }
    return i;
}
Only trick thing about reading LLVM is thinking in SSA.

```llvm
define i32 @count(i32 %n) {
  entry:
    br label %loop
  loop:
    %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
    %nextvar = add i32 %i, 1
    %cmptmp = icmp ult i32 %i, %n
    %booltmp = zext i1 %cmptmp to i32
    %loopcond = icmp ne i32 %booltmp, 0
    br i1 %loopcond, label %loop, label %afterloop
  afterloop:
    ret i32 %i
}
```
entry:
  br label %loop

loop:
  %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
  %nextvar = add i32 %i, 1
  %cmptmp = icmp ult i32 %i, %n
  %booltmp = zext i1 %cmptmp to i32
  %loopcond = icmp ne i32 %booltmp, 0
  br i1 %loopcond, label %loop, label %afterloop

T F

afterloop:
  ret i32 %i
Julia

```
function sinc(x)
    if x == 0
        return 1.0
    else
        return sin(x*pi) / (x*pi)
    end
end
```
julia> code_llvm(sincf, (Int,))
define double @julia_sincf(i64) {
  top:
    %1 = icmp eq i64 %0, 0
    br i1 %1, label %if, label %L
  if:
    ret double 1.000000e+00
  L:
    %2 = sitofp i64 %0 to double
    %3 = fmul double %2, 3.14159265359e+00
    %4 = call double @sin(double %3)
    %5 = fcmp uno double %3, 0.000000e+00
    %6 = fcmp ord double %4, 0.000000e+00
    %7 = or i1 %6, %5
    br i1 %7, label %pass, label %fail
  pass:
    %9 = fdiv double %4, %3
    ret double %9
  fail:

julia> code_llvm(sincf, (Float64,))
define double @julia_sincf(double) {
   top:
      %1 = fcmp une double %0, 0.000000e+00
      br i1 %1, label %L, label %if
    if:
      ret double 1.000000e+00
    L:
      %2 = fmul double %0, 3.14159265359e+00
      %3 = call double @sin(double %2)
      %4 = fcmp uno double %2, 0.000000e+00
      %5 = fcmp ord double %3, 0.000000e+00
      %6 = or i1 %5, %4
      br i1 %6, label %pass, label %fail
    pass:
      %8 = fdiv double %3, %2
      ret double %8
    fail:
      ...

Harvard-Smithsonian Center for Astrophysics  LLVM Optimized Python for Scientific Computing
That’s pretty straightforward, Can we do the same in Python?

```python
import llvm.ee as le
import llvm.core as lc

mod = lc.Module.new('my_llvmpy_module')

int_type = lc.Type.int()

func_type = lc.Type.function(int_type, argtypes=[], False)
lfunc = lc.Function.new(module, func_type, "main")
```
# Interactively build the LLVM module
entry_block = lfunc.append_basic_block("entry")
builder = lc.Builder.new(entry_block)

value = lc.Constant.int(int_type, 42)
block = builder.ret(value)
In[1]: print mod

; ModuleID = 'my_llvmpy_module'

define i32 @main() {
  entry:
    ret i32 42
}
# Build the JIT Engine

tm = le.TargetMachine.new(features='', cm=le.CM_JITDEFAULT)

eb = le.EngineBuilder.new(mod)

jit = eb.create(tm)
In[2]: ret = jit.run_function(fn, [])
In[3]: print ret.as_int()
42
In[4]: print mod.to_native_assembly()

.file "my_llvmpy_module"
.text
.globl main
.align 16, 0x90
.type main,@function
main:
   .cfi_startproc
   movl $42, %eax
   ret
.Ltmp0:
   .size main, .Ltmp0-main
   .cfi_endproc

.section " .note.GNU-stack",",",@progbits
Selective Embedded Just-In-Time Specialization (SEJITS)

Appealing for two reasons:

- Short term: Speeding up existing logic with a single decorator.
- Long term: Not have to rewrite existing logic in another language.
Python is great for rapid development of mathematical models and high-level thinking.

You can state your equation or model in a few lines of Python, but to rewrite it in efficient vectorized NumPy you need to piece together 15 ufuncs and create two deep copies. When you run the model on $10^{11}$ inputs it takes a week!
@autojit
def f(x, n, m):
    y = 1
    z = x
    while n > 0:
        if n & 1:
            y = (y * z) % m
            z = (z * z) % m
        n //= 2
    return y
```
import numpy as np

a = np.random.random_sample((size,))
b = np.random.random_sample((size,))
r = np.dot(a, b)

What if we could write np.dot in Python instead of C and suffer no loss in performance?
```
Our seemingly pseudocode is as often in the same order as compiled Fortran kernels, but with fast prototyping.

```python
@autojit
def dot(a, b):
    c = 0
    n = a.shape[0]
    for i in xrange(n):
        c += a[i] * b[i]
    return c

a = np.random.random_sample((size,))
b = np.random.random_sample((size,))
r = dot(a, b)
```

Python : 2.717 s  
Numpy : 1.331 ms  
Numba : 1.322 ms
from PIL import Image
from scipy.ndimage import convolve
from matplotlib.pyplot import imshow, show

ifile = Image.open('pillars.jpg')
image = np.asarray(ifile, dtype=np.float32)

# 5 x 5 High Pass Filter
filter = np.array(
    [[[-1, -1, -1, -1, -1],
      [-1, 1, 2, 1, -1],
      [-1, 2, 4, 2, -1],
      [-1, 1, 2, 1, -1],
      [-1, -1, -1, -1, -1]]
)

result = convolve(image, filter)
imshow(result)
@autojit
def filter(image, filt, output):
    M, N = image.shape
    m, n = filt.shape
    for i in range(m//2, M-m//2):
        for j in range(n//2, N-n//2):
            result = 0.0
            for k in range(m):
                for l in range(n):
                    result += image[i+k-m//2,j+l-n//2]*filt[k, l]
            output[i, j] = result

output = image.copy()
filter(image, filt, output)
imshow(output)
On top of being much easier to use (i.e. automatic type inference by autojit) it’s now about 50% faster, and is even a few percent faster than the Cython option. […] I’m becoming more and more convinced that Numba is the future of fast scientific computing in Python.
```python
@autojit
def dilate(x, k):
    m,n = x.shape
    y = np.empty_like(x)
    for i in xrange(m):
        for j in xrange(n):
            currmax = x[i,j]
            for ii in xrange(max(0, i-k/2), min(m, i+k/2+1)):
                for jj in xrange(max(0, j-k/2), min(n, j+k/2+1)):
                    elt = x[ii,jj]
                    if elt > currmax:
                        currmax = elt
            y[i,j] = currmax
    return y
```
PDE solver for 2D Wave Equation

Wave Equation
Caveats

Numba/LLVM isn’t a black box.

- Subset of Python
- Closures
- Classes
- Exceptions
- Generators
- Arbitrary data structures
- Most of Python will not work
- Different semantics than Python
- **nopython** - Types are unboxed, logic does not go through CPython runtime.
- **python** - Logic goes through the CPython runtime.
- **autojit** - Inferred types from usage.
- **jit** - Manually explicit types
Optimizations

- Generally state of the art.
- Numerically stable
- Perils of auto-optimizers:
  - Relies on a lot of heuristics and pattern matching for detecting common patterns.
  - Optimizations are often in cascade, one optimization opens up opportunities and so on.
  - When the cascade fails (simple transposition) the performance can vary by orders of magnitude. Can be difficult to debug unless you grok the LLVM internals.
Questions / Comments?
“What is the simplest possible thing we can implement that will be useful?”

http://dev.stephendiehl.com/numpile

Let’s write a simple numeric specializer in 1 Python module, 1000 LOC. Will be published on the internet. Doesn’t fit in 30 minute talk.

1. Decorate the function.
2. Grab the AST.
3. Introspect the type of arguments.
4. Do type inference.
5. Compile function.
6. For array arguments, grab pointer to unboxed memory.
7. For scalar arguments, grab C value from PyObject.
Resources

- Mapping High Level Constructs to LLVM IR
- Implementing a JIT Compiled Language with Haskell and LLVM
- The need for an embedded array expression compiler for NumPy
- llvmlite