PyCon2014 China-Hang Zhou

Python & LLVM

单击添加文字

李枫
hkli2013@126.com
2014.11.22

Content

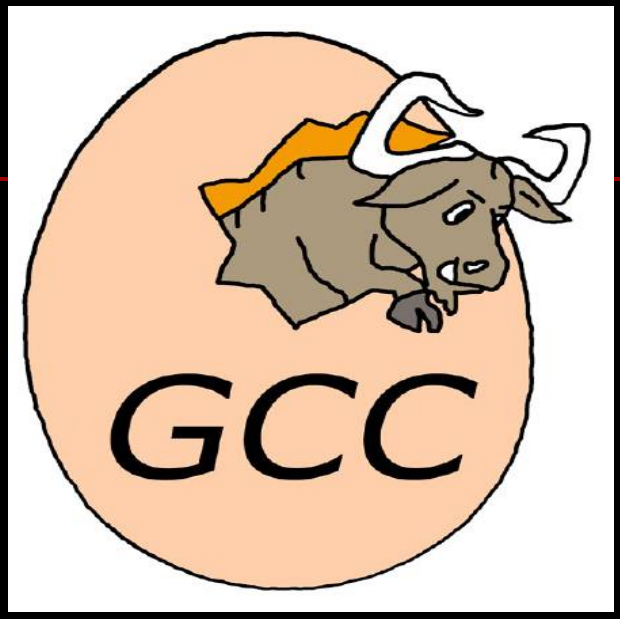
I. Introduction

1) LLVM

- Vikram Adve
  - started in 2000 at the University of Illinois
- Chris Lattner
  - work for Apple since 2005

- Compiler Infrastructure
- LLVM IR (Intermediate Representation)
- Clang Front-end
- LLVM JIT
- Modular Design
- Sanitizers
- Better Diagnostics

GCC & LLVM



ld.bfd / ld.gold	lld / mclinker
gdb	lldb
as/objdump	MC layer in LLVM
libstdc++	libc++
libgcc	libcompiler-rt

Benchmarks

GCC 5.0 Snapshot Compiler Benchmark Fedora 21:

SYSTEM HARDWARE / SOFTWARE

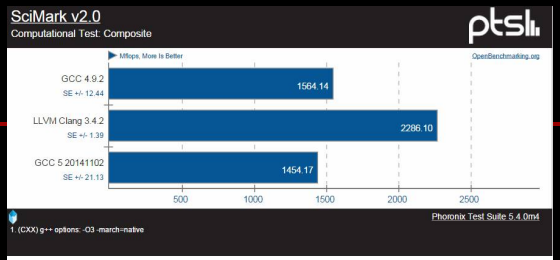
GCC 5.0 Snapshot Compiler Benchmark Fedora 21

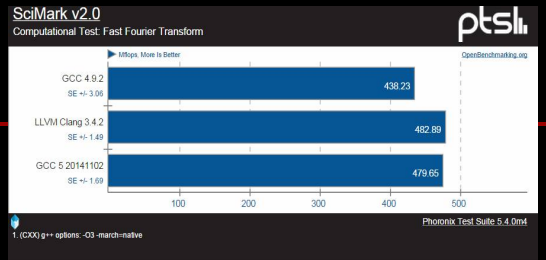
OpenBenchmarking.org	Phoronix Test Suite 5.2.0
Intel Xeon E5-1680 v3 @ 3.80GHz (16 Cores)	Processor
MSI X99S_SLI PLUS (MS-7885) v1.0	Motherboard
Intel Xeon E5 v3/Core	Chipset
16384MB	Memory
80GB INTEL SSDSCKGW08	Disk
eVGA NVIDIA GeForce GT 740 1024MB	Graphics
Intel C610/X99	Audio
VE228	Monitor
Intel Connection	Network
Fedora 21	OS
3.18.0-0.rc3.git2.2.fc22.1.x86_64 (x86_64)	Kernel
GNOME Shell 3.14.1.5	Desktop
X Server 1.16.1	Display Server
nouveau 1.0.10	Display Driver
3.3 Mesa 10.3.2 Gallium 0.4	OpenGL
GCC 4.9.2 20141101 + Clang 3.4.2 + LLVM 3.4.2 Clang 3.4.2 + LLVM 3.4.2 GCC 5.0.0 20141102 + LLVM 3.4.2	Compiler
ext4	File-System
1920x1080	Screen Resolution

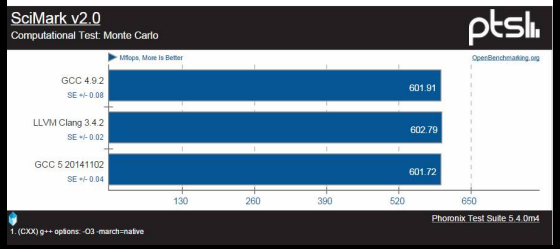
GCC 5.0 Snapshot Compiler Benchmark Fedora 21 Performance

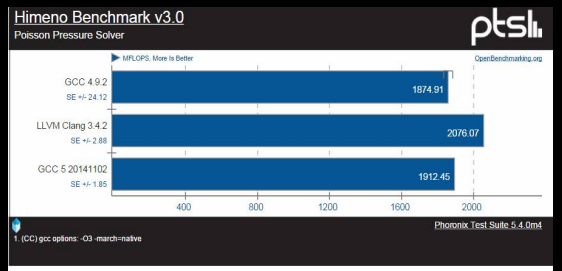
- GCC 4.9.2: --build=x86_64-redhat-linux --disable-libgcj --disable-libunwind-exceptions --enable-__cxa_atexit --enable-bootstrap --enable-checking=release --enable-gnu-indirect-function --enable-gnu-unique-object --enable-initfini-array --enable-languages=c,c++,objc,obj-c++,fortran,ada,go,lto --enable-multilib --enable-plugin --enable-shared --enable-threads=posix --mandir=/usr/share/man --with-arch_32=i686 --with-linker-hash-style=gnu --with-tune=generic
- LLVM Clang 3.4.2: Optimized build; Built Aug 14 2014 (16:12:44); Default target: x86_64-redhat-linux-gnu; Host CPU: core-avx2
- GCC 5 20141102: --disable-multilib --enable-checking=release --enable-languages=c,c++
- Scaling Governor: intel_pstate powersave

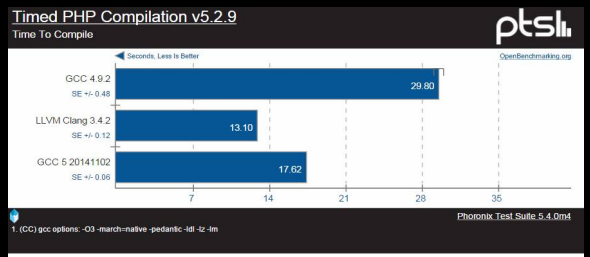
System Logs
OPC Classification



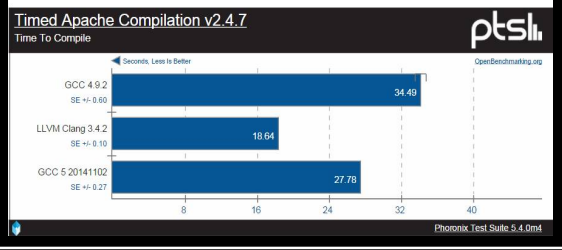


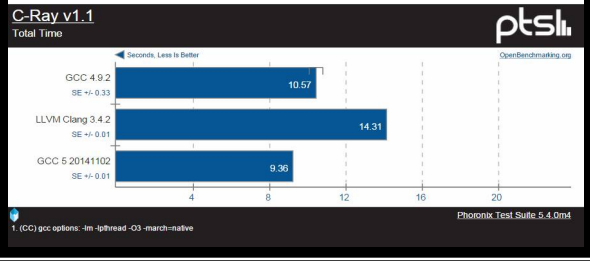






PACHE COMPILATION





Phoronix Test Suite	GCC 4.9.2	LLVM Clang 3.4.2	GCC 5 20141102
SciMark	1564.14	2286.10	1454.17
SciMark	601.91	602.79	601.72
SciMark	438.23	482.89	479.65
SciMark	2711.88	3091.20	2305.93
SciMark	2865.36	5617.70	2737.23
SciMark	1203.33	1635.93	1146.33
Himeno Benchmark	1874.91	2076.07	1912.45
Timed Apache Compilation	34.49	18.64	27.78
Timed PHP Compilation	29.80	13.10	17.62
C-Ray	10.57	14.31	9.36
Smallpt	17	135	14
Bullet Physics Engine	5.15	5.34	5.00
Bullet Physics Engine	5.44	5.88	5.48
Bullet Physics Engine	5.14	5.72	5.11
Bullet Physics Engine	3.42	3.44	3.33
Bullet Physics Engine	1.12	1.13	1.12
Bullet Physics Engine	1.33	1.37	1.34
FLAC Audio Encoding	3.89	4.21	4.08
LAME MP3 Encoding	11.36	11.62	11.69
Apache Benchmark	14484.47	13513.76	13480.80

OpenBenchmarking.org

II. LLVMPY

1) Overview

llvm.org/releases/3.1/docs/ReleaseNotes.html

Python Bindings

Officially supported Python bindings have been added! Feature support is far from complete. The current bindings support interfaces to:

- Object File Interface
- Disassembler

Using the Object File Interface, it is possible to inspect binary object files. Think of it as a Python version of readelf or llvm-objdump.

Support for additional features is currently being developed by community contributors. If you are interested in shaping the direction of the Python bindings, please express your intent on IRC or the developers list.

llvmpy is a Python wrapper around the llvm C++ library which allows simple access to compiler tools. It can be used for a lot of things, but here are some ideas:

- dynamically create LLVM IR for linking with LLVM IR produced by CLANG or dragonegg
- build machine code dynamically using LLVM execution engine
- use together with PLY or other tokenizer and parser to write a complete compiler in Python

LLVMPY

http://cloc.sourceforge.net v 1.60 T=0.82 s (290.0 files/s, 56329.5 lines/s)

Language	files	blank	comment	code
Python	187	4479	1910	16801
HTML	21	2936	304	16441
C/C++ Header	8	219	111	1385
C++	1	89	20	544
CSS	4	50	9	346
make	4	57	5	204
C	7	36	34	165
Pascal	1	31	7	87
Javascript	1	5	15	47
YAML	2	12	5	47
Bourne Shell	2	3	2	9
DOS Batch	1	0	0	6
SUM:	239	7917	2422	36082

LLVM

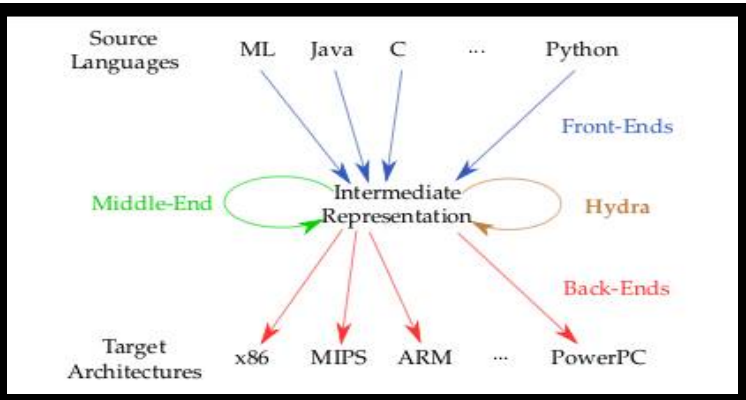
Language	files	blank	comment	code
C++	6467	295272	368702	1624436
C/C++ Header	3234	101767	173954	449106
HTML	501	15409	4304	269532
C	2879	32549	90228	206437
Assembly	1304	29303	60447	113872
Python	693	20618	22336	75695
Objective C	1251	13386	57703	47129
Bourne Shell	60	2911	2373	23494
Objective C++	334	4484	23001	17441
CMake	343	1825	1448	12575
Teamcenter def	32	931	452	6453
OCaml	70	1743	2739	5648
Perl	21	938	865	5619
make	355	1715	2640	4875
Pascal	13	1137	5864	3753
Go	21	384	585	2876

Architecture

- **The llvmpy is a Python package, consisting of 6 modules, that wrap over enough LLVM APIs to allow the implementation of your own compiler/VM backend in pure Python**

Out of the 6 modules, one is an "extension" module (i.e., it is written in C), and another one is a small private utility module, which leaves 4 public modules. These are:

- *llvm* – top-level package, common classes (like exceptions)
- *llvm.core* – IR-related APIs
- *llvm.ee* – execution engine related APIs
- *llvm.passes* – pass manager and passes related APIs



Native Bindings for Python

SWIG(LLDB…) **ctypes(CPython…)** **Cython(Sage…)** **CFFI/CPPYY(PyPy…)**

2) Example

- http://www.llvmpy.org/llvmpy-doc/dev/doc/firstexample.html

Let's create a (LLVM) module containing a single function, corresponding to the `c` function:

```c
int sum(int a, int b)
{
    return a + b;
}
```

Here's how it looks in llvmpy:

```
#!/usr/bin/env python

# Import the llvmpy modules.
from llvm import *
from llvm.core import *

# Create an (empty) module.
my_module = Module.new('my_module')

# All the types involved here are "int"s. This type is represented
# by an object of the llvm.core.Type class:
ty_int = Type.int()     # by default 32 bits

# We need to represent the class of functions that accept two integers
# and return an integer. This is represented by an object of the
# function type (llvm.core.FunctionType):
ty_func = Type.function(ty_int, [ty_int, ty_int])

# Now we need a function named 'sum' of this type. Functions are not
# free-standing (in llvmpy); it needs to be contained in a module.

f_sum = my_module.add_function(ty_func, "sum")

# Let's name the function arguments as 'a' and 'b'.
f_sum.args[0].name = "a"
f_sum.args[1].name = "b"

# Our function needs a "basic block" -- a set of instructions that
# end with a terminator (like return, branch etc.). By convention
# the first block is called "entry".
bb = f_sum.append_basic_block("entry")

# Let's add instructions into the block. For this, we need an
# instruction builder:
builder = Builder.new(bb)

# OK, now for the instructions themselves. We'll create an add
# instruction that returns the sum as a value, which we'll use
# a ret instruction to return.
tmp = builder.add(f_sum.args[0], f_sum.args[1], "tmp")
builder.ret(tmp)

# We've completed the definition now! Let's see the LLVM assembly
# language representation of what we've created:

print my_module
```

Here is the output:

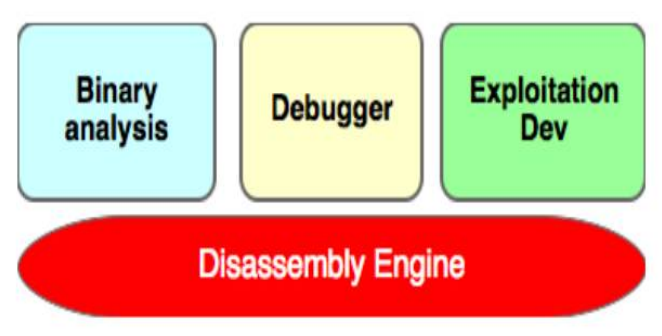
```
; ModuleID = 'my_module'

define i32 @sum(i32 %a, i32 %b) {
entry:
        %tmp = add i32 %a, %b           ; <i32> [#uses=1]
        ret i32 %tmp
}
```

```
clang -emit-llvm -O1 -S sum.c
```

III. Assembly

1) Capstone

- **Next-Gen Disassembly Framework**



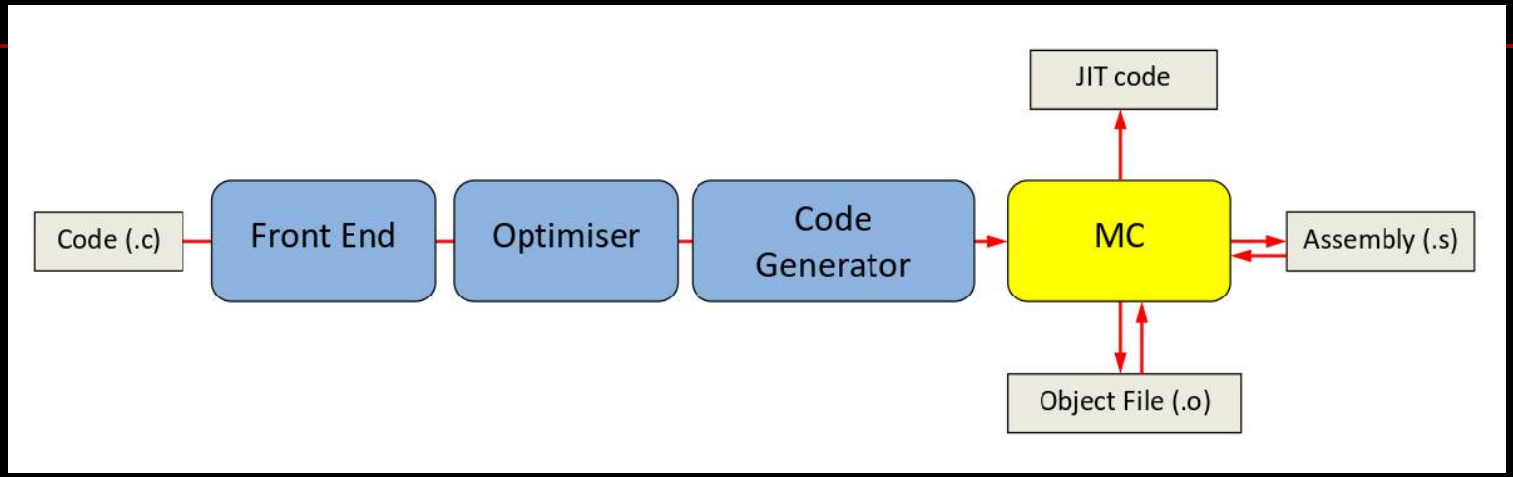
- Multi-arch: X86 + Arm + Arm64 + Mips + PPC (surpassed eventually)
- Multi-platform: Windows + MacOSX + Linux (surpassed eventually).
- Multi-bindings: Python + Ruby + Java + C# (surpassed eventually).
- Clean, simple, intuitive & architecture-neutral API.
- Provide break-down details on instructions.
- Friendly license: BSD.

Features	Distorm3	BeaEngine	Udis86	Libopcode
X86|Arm	✓|X	✓|X	✓|X	✓|✓ [1]
Linux|Windows	✓|✓	✓|✓	✓|✓	✓|X
Python|Ruby bindings	✓|X [2]	✓|X	✓|X	✓|X
Update	X	?	X	X
License	GPL	LGPL3	BSD	GPL

- Support Microsoft Visual Studio (so Windows native compilation using *MSVC* is possible).
- Support CMake compilation.
- Cross-compile for *Android*.
- Build libraries/tests using XCode project
- Much faster, while consuming less memory for all architectures.

2) LLVM's Machine Code (MC) layer

- a new framework that was designed to aid in the creation of LLVM-based assemblers and disassemblers



- Capstone framework is based on the MC component of the LLVM compiler infrastructure, but is superior to LLVM's disassembler:

  http://www.capstone-engine.org/beyond_llvm.html

3) Python binding

- http://capstone-engine.org/lang_python.html
- Support both Python 3 and Python 2

The sample below shows how to extract the details on instruction operands of ARM64 code.

```
1  from capstone import *
2  from capstone.arm64 import *
3
4  CODE = "\xe1\x0b\x40\xb9\x20\x04\x81\xda\x20\x08\x02\x8b"
5
6  md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
7  md.detail = True
8
9  for insn in md.disasm(CODE, 0x38):
10     print("0x%x:\t%s\t%s" %(insn.address, insn.mnemonic, insn.op_str))
11
12     if len(insn.operands) > 0:
13         print("\tNumber of operands: %u" %len(insn.operands))
14         c = -1
15         for i in insn.operands:
16             c += 1
17             if i.type == ARM64_OP_REG:
18                 print("\t\toperands[%u].type: REG = %s" %(c, insn.reg_name(i.value.reg)))
19             if i.type == ARM64_OP_IMM:
20                 print("\t\toperands[%u].type: IMM = 0x%x" %(c, i.value.imm))
21             if i.type == ARM64_OP_CIMM:
22                 print("\t\toperands[%u].type: C-IMM = %u" %(c, i.value.imm))
23             if i.type == ARM64_OP_FP:
24                 print("\t\toperands[%u].type: FP = %f" %(c, i.value.fp))
25             if i.type == ARM64_OP_MEM:
26                 print("\t\toperands[%u].type: MEM" %c)
27                 if i.value.mem.base != 0:
28                     print("\t\t\toperands[%u].mem.base: REG = %s" \
29                         %(c, insn.reg_name(i.value.mem.base)))
30                 if i.value.mem.index != 0:
31                     print("\t\t\toperands[%u].mem.index: REG = %s" \
32                         %(c, insn.reg_name(i.value.mem.index)))
33                 if i.value.mem.disp != 0:
34                     print("\t\t\toperands[%u].mem.disp: 0x%x" \
35                         %(c, i.value.mem.disp))
36
37             if i.shift.type != ARM64_SFT_INVALID and i.shift.value:
38                 print("\t\t\tShift: type = %u, value = %u" \
39                     %(i.shift.type, i.shift.value))
40
41             if i.ext != ARM64_EXT_INVALID:
42                 print("\t\t\tExt: %u" %i.ext)
43
44     if insn.writeback:
45         print("\tWrite-back: True")
46     if not insn.cc in [ARM64_CC_AL, ARM64_CC_INVALID]:
47         print("\tCode condition: %u" %insn.cc)
48     if insn.update_flags:
49         print("\tUpdate-flags: True")
```

- Line 12: Check if this instruction has any operands to print out.
- Line 17 ~ 18: If this operand is register (reflected by type *ARM64_OP_REG*), then print out its register name.
- Line 19 ~ 20: If this operand is immediate (reflected by type *ARM64_OP_IMM*), then print out its numerical value.
- Line 21 ~ 22: If this operand is of type C-IMM (coprocessor register type, reflected by *ARM64_OP_CIMM*), then print out its index value.
- Line 23 ~ 24: If this operand is real number (reflected by type *ARM64_OP_FP*), then print out its numerical value.
- Line 25 ~ 35: If this operand is memory reference (reflected by type *ARM64_OP_MEM*), then print out its base/index registers, together with offset value.
- Line 37 ~ 42: If this operand uses shift or extender, print out their value.
- Line 44 ~ 45: If this instruction writes back its value afterwards, print out that.
- Line 46 ~ 47: Print out the code condition of this instruction.
- Line 48 ~ 49: If this instruction update flags, print out that.

The output of the above sample is like below.

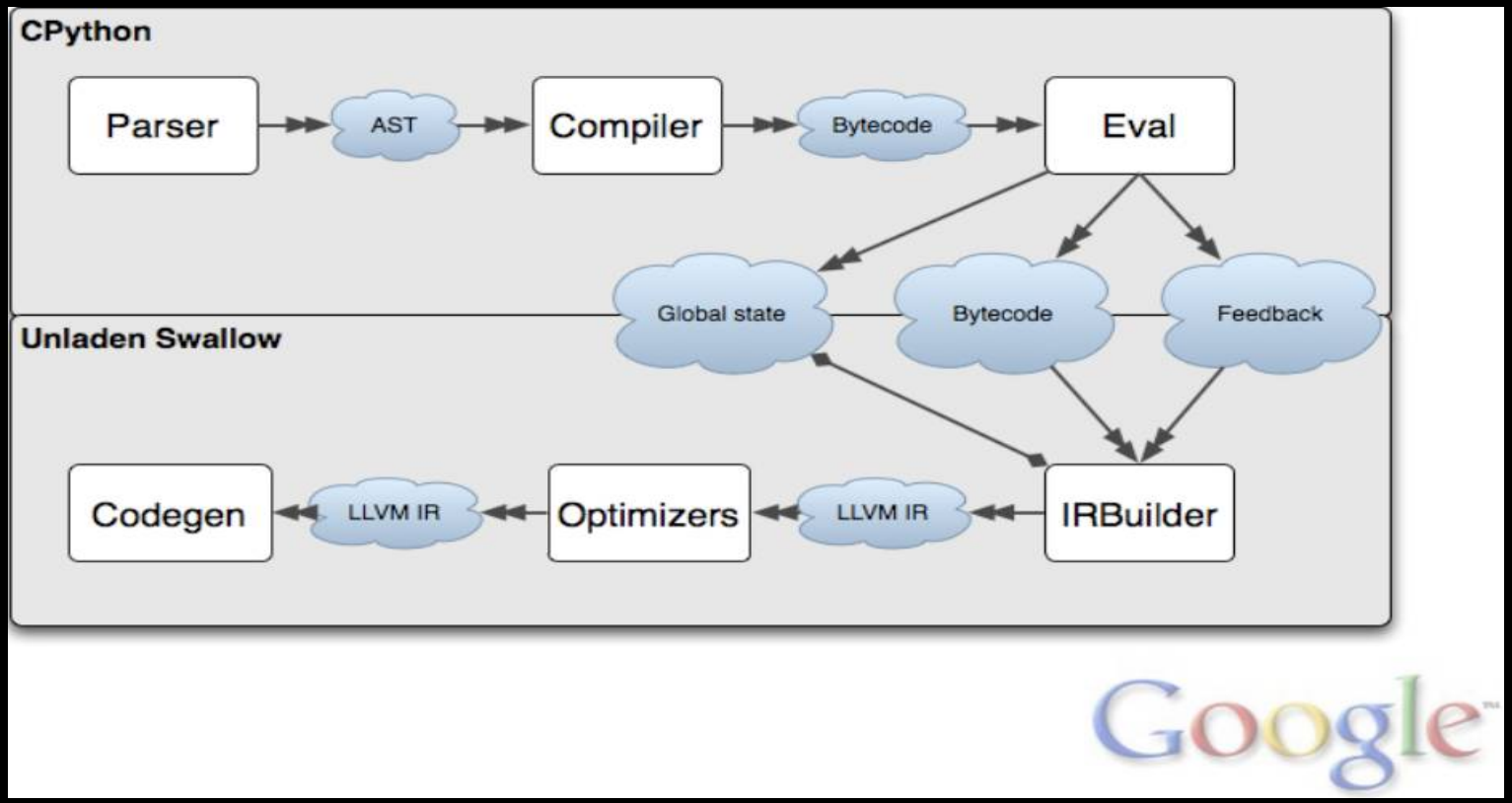
```
0x38:   ldr     w1, [sp, #8]
        Number of operands: 2
                operands[0].type: REG = w1
                operands[1].type: MEM
                        operands[1].mem.base: REG = sp
                        operands[1].mem.disp: 0x8

0x3c:   csneg   x0, x1, x1, eq
        Number of operands: 3
                operands[0].type: REG = x0
                operands[1].type: REG = x1
                operands[2].type: REG = x1
        Code condition: 1

0x40:   add     x0, x1, x2, lsl #2
        Number of operands: 3
                operands[0].type: REG = x0
                operands[1].type: REG = x1
                operands[2].type: REG = x2
                        Shift: type = 1, value = 2
```

IV. Use Case

1) LLVM-based Python implementation

Unladen Swallow

- open source project sponsored by Google

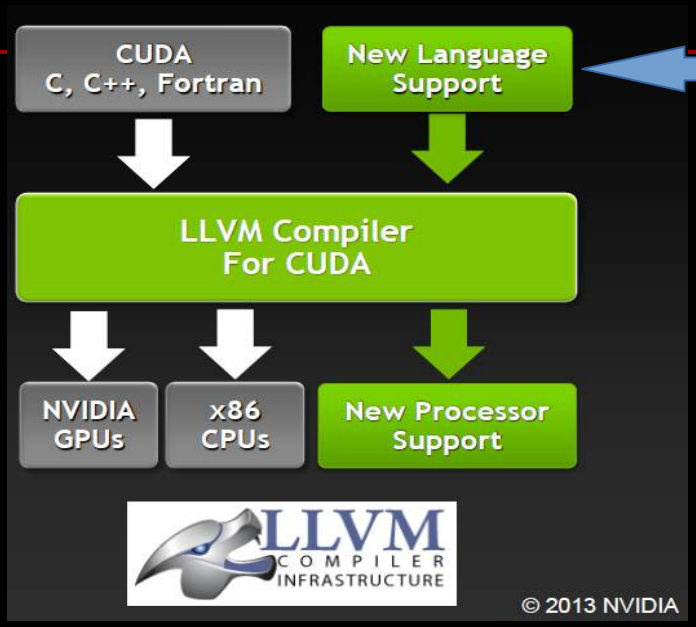


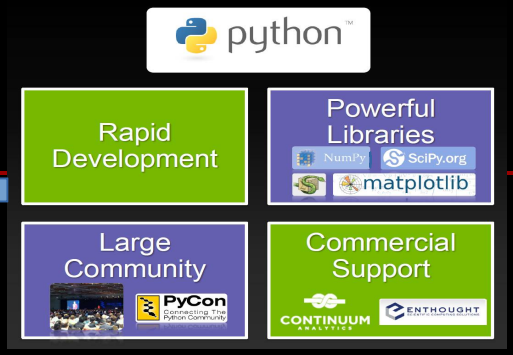
Pyston

- open source project sponsored by Dropbox
- a new, under-development Python implementation built using LLVM and modern JIT techniques with the goal of achieving good performance
- only targets Python 2.7, and runs on x86_64 platforms currently

2) HPC

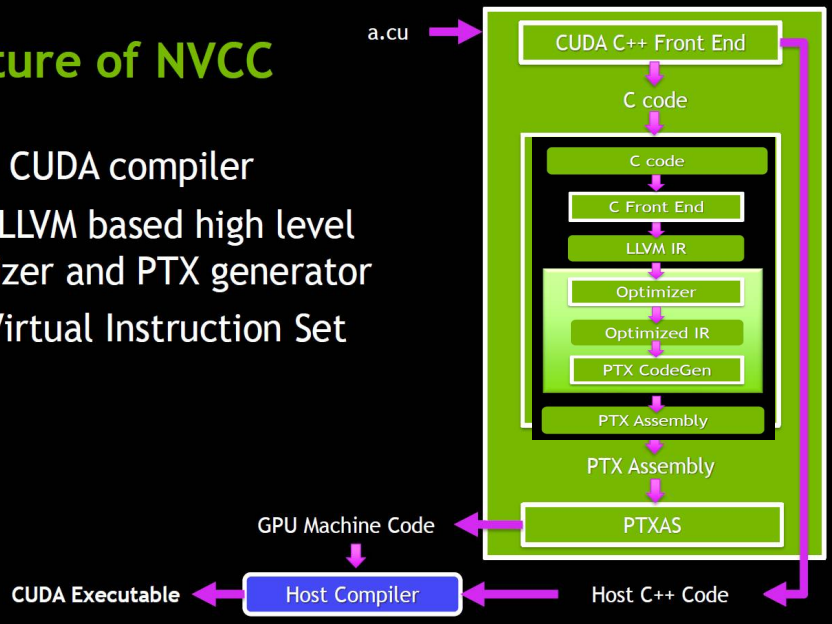
- Heterogeneous Parallel Computing
- High Performance Computing





Structure of NVCC

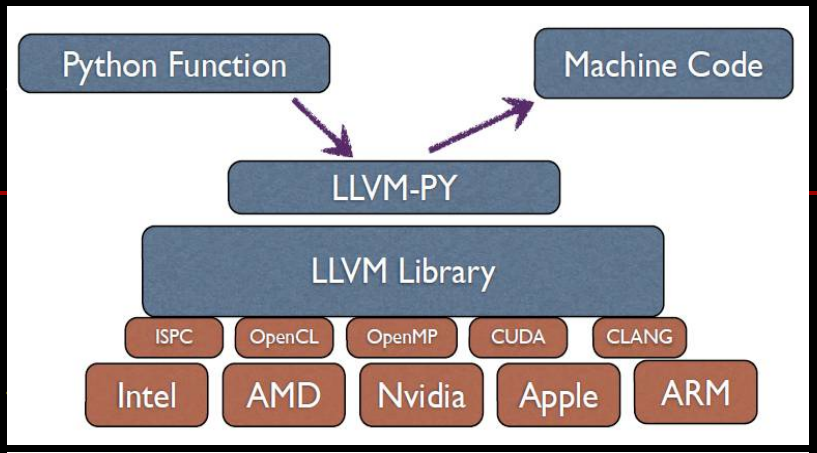
- NVCC: CUDA compiler
- CICC: LLVM based high level optimizer and PTX generator
- PTX: Virtual Instruction Set



Goals for the CUDA Platform

Simplicity	• Learn, adopt, & use parallelism with ease
Productivity	• Quickly achieve feature & performance goals
Portability	• Write code that can execute on all targets
Performance	• High absolute performance and scalability

NumbaPro



Black Scholes example

https://github.com/ContinuumIO/numbapro-examples/tree/master/blackscholes

```
$ conda update conda
$ conda create -n gpu accelerate
$ source activate gpu
```

Windows

```
$ activate gpu
```

Program	Time (ms)	Speed-up
NumPy	1707.38	1x
Numba	786.84	2.17x
CUDA Python	85.24	20.03x

Getting Started

Let's start with a simple function to add together all the pairwise values in two NumPy arrays. Asking NumbaPro to compile this Python function to vectorized machine code for execution on the CPU is as simple as adding a single line of code (invoked via a decorator on the function):

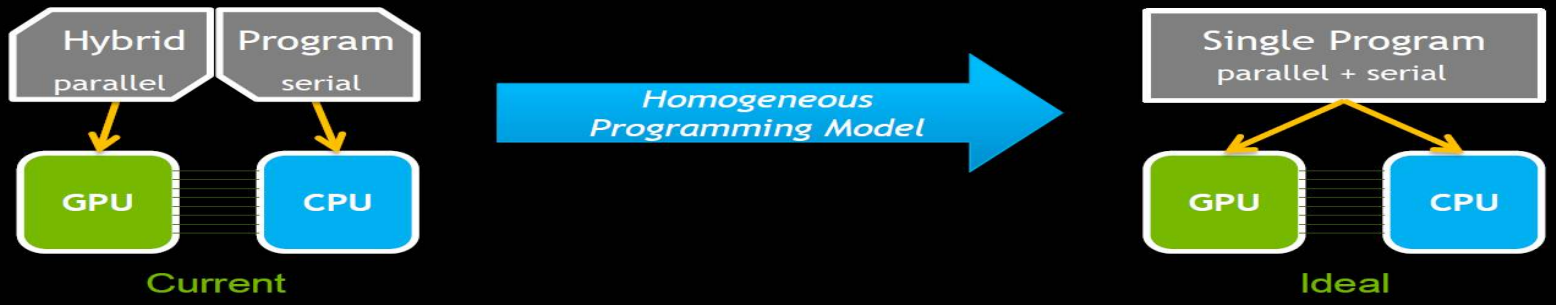
```
from numbapro import vectorize

@vectorize(['float32(float32, float32)'], target='cpu')
def sum(a, b):
    return a + b
```

Similarly, one can instead target the GPU for execution of the same Python function by modifying a single line in the above example:

```
@vectorize(['float32(float32, float32)'], target='gpu')
```

Targeting the GPU for execution introduces the potential for numerous GPU-specific optimizations so as a starting point for more complex scenarios, one can also target the GPU with NumbaPro via its Just-In-Time (JIT) compiler:

```
from numbapro import cuda

@cuda.jit('void(float32[:], float32[:], float32[:])')
def sum(a, b, result):
    i = cuda.grid(1)   # equals to threadIdx.x + blockIdx.x * blockDim.x
    result[i] = a[i] + b[i]

# Invoke like:  sum[grid_dim, block_dim](big_input_1, big_input_2, result_array)
```



3) The future

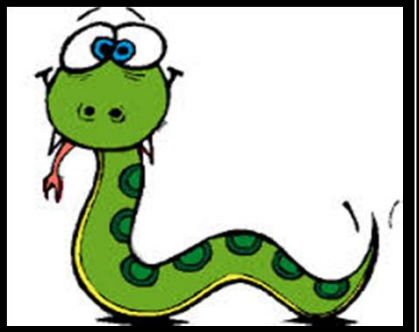
V. Reference

- http://en.wikipedia.org/wiki/Wiki
- https://gcc.gnu.org
- http://llvm.org
- http://www.llvmpy.org/
- http://swig.org
- https://docs.python.org/3/library/ctypes.html
- http://cython.org
- https://cffi.readthedocs.org/
- http://pypy.readthedocs.org/en/latest/cppyy.html
- https://developer.nvidia.com/CUDA-LLVM-Compiler
- http://docs.continuum.io/numbapro/index.html
- http://en.wikipedia.org/wiki/Unladen_Swallow
- https://github.com/dropbox/pyston
- http://lldb.llvm.org/
- http://llvm.linuxfoundation.org

Q & A

单击添加文字

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```



Thanks!