Numba: A LLVM-Based Python JIT Compiler

김동효

Content

Introduction [1/3]

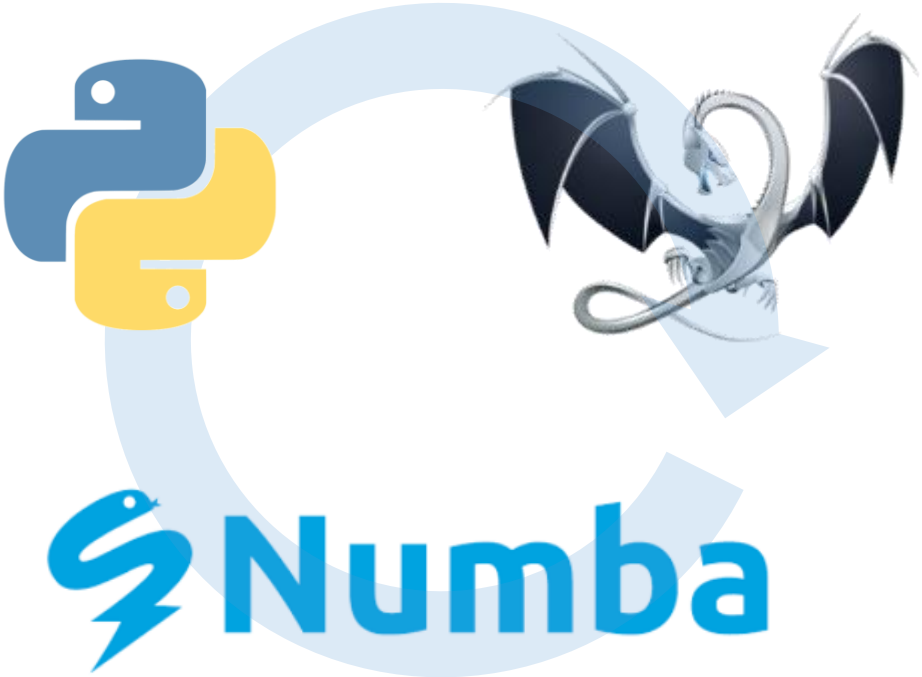
- **Python**
  - python은 간단한 syntax 및 유연한 semantics, dynamic typing, 다양한 라이브러리로 인해 생산성 높은 언어로 간주되고 있음
  - 소스코드를 한번에 한 줄씩 읽어 들여 실행하는 인터프리터 언어로, cpython 인터프리터를 이용하여 python 소스코드의 컴파일을 수행함
  - 하지만, cpython은 인터프리터로 동작하는 스크립트 언어이기에 실행 속도는 컴파일 언어 보다 느리다는 단점이 있음

Introduction [2/3]

- **Numpy(Numerical Python)**
  - **Python기반 데이터 분석 환경에서 수치 및 과학 계산을 위한 라이브러리**
  - **다차원 배열 및 행렬 연산에 필요한 다양한 함수를 지원하며 수치 계산의 기초가 되는 다차원 배열 ndarray 객체를 제공함**
  - **Ndarray와 함께 python operator를 사용할 경우, 인터프리터에 큰 속도를 제공하지만 python의 element-wise를 루프에서 반복하는 것은 실행 속도 측면에서 비 효율적임**

Introduction [3/3]

- 본 논문에서는 python 환경에서 numpy 배열 및 루프의 반복을 이용하여 수치 계산을 수행할 때 발생하는 실행 속도 저하 문제를 개선하기 위한 LLVM 기반 JIT 컴파일러인 “Numba” 를 소개함



A JIT for Numeric Python [1/10]

- **JIT Compilation(Just-In-Time Compilation)**
  - 프로그램을 실제 실행하는 시점에 기계어로 컴파일하여 실행 속도를 향상시키는 방법
  - 즉, 실행 시점에 기계어 코드를 생성하여 캐싱하고, 동일한 함수가 여러 번 호출될 때 마다 캐싱된 코드를 불러와 매번 기계어 코드를 생성하는 것을 방지하여 실행 속도를 향상 시킬 수 있음

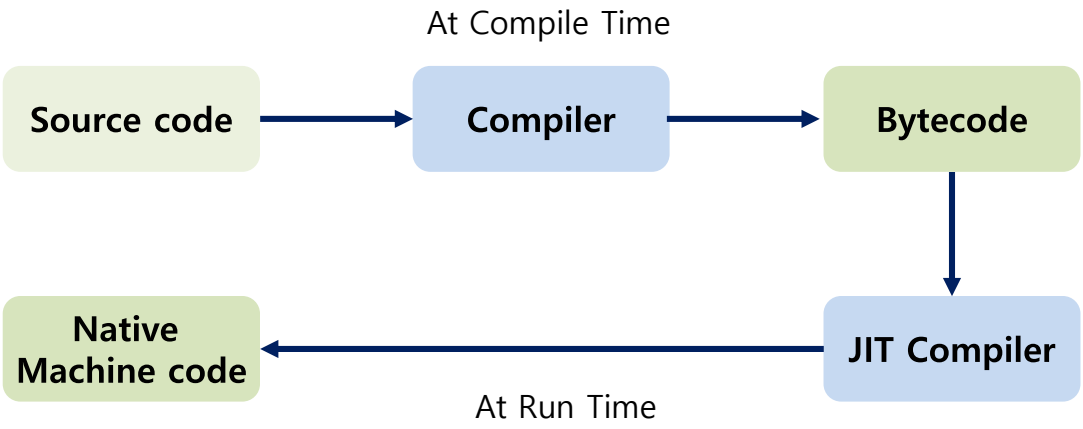


Fig. 2. JIT Compilation

A JIT for Numeric Python [2/10]

- **Numba**
  - CPython을 위한 LLVM 기반 function-at-a-time JIT 컴파일러 라이브러리
  - 런타임 시 Python 함수를 최적화된 기계어 코드로 변환하여 빠른 속도 제공
  - Single-threaded CPU, Multi-threaded CPU, GPU 환경 지원
  - Numpy 배열 및 함수들과 함께 사용하도록 설계되어 있으며 Numba를 이용하여 수치연산을 수행할 경우, C 또는 FORTRAN의 속도와 유사하게 컴파일 할 수 있는 장점이 있음

OS	HW	SW
Windows (7 and later)	32 and 64-bit CPUs (Incl Xeon Phi)	Python 2.7, 3.4-3.6
OS X (10.9 and later)	NVIDIA GPUs	NumPy 1.10 and later
Linux (RHEL 6 and later)	Experimental support for ARM, POWER8/9 and AMD GPUs (ROCm)	

Fig. 3. Numba 지원 플랫폼 및 하드웨어

Matrix Size	Numba	C
64 x 64	463x	453x
128 x 128	454x	407x
256 x 256	280x	263x
512 x 512	276x	268x

Table. 1. 행렬 곱 연산에 따른 Numba 및 C 실행속도 향상률(비교 대상:Cpython)

A JIT for Numeric Python [3/10]

- **Numba Test (1/5)**
  - ➢ 함수에 python jit decorator (@jit)를 wrapping하여 사용

```python
from numba import jit
from numpy import arange

@jit #(nopython = True, cache= True)
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

Fig. 4. jit decorator 선언

➢ @jit 의 옵션

- ✓ python object 모드(default): 모든 값을 python 객체로 처리하고 python C API를 사용하여 컴파일을 수행
- ✓ nopython 모드: Decorating 된 함수를 컴파일하여 인터프리터 개입 없이 실행 (C API에 액세스 하지 않고 컴파일하는 방식)
- ✓ parallel 모드: 자동 병렬화 처리 활성화
- ✓ cache 모드: Python 프로그램을 호출할 때 마다 컴파일을 방지하기 위해 캐시에 쓰도록 지시

A JIT for Numeric Python [4/10]

- **Numba Test (2/5)**

```python
def f_py(I, J):
    res = 0
    for i in range(I):
        for j in range (J):
            res += int(cos(log(1)))
    return res

def f_np(I, J):
    a = np.ones((I, J), dtype=np.float64)
    return int(np.sum(np.cos(np.log(a)))), a

I, J = 10000, 10000

# Python
start = time.time()
f_py(I, J)
print("Python time : " + str(time.time() - start) + " " + "seconds")

# Numpy
start = time.time()
res, a = f_np(I, J)
print("Numpy time : " + str(time.time() - start) + " " + "seconds")

# Numba
start = time.time()
f_nb = nb.jit(f_py)
f_nb(I, J)
print("Numba time : " + str(time.time() - start) + " " + "seconds")
```

```
Python time : 39.9320001602 seconds
Numpy time : 1.78499984741 seconds
Numba time : 0.171000003815 seconds
```

Fig. 5. jit decorator를 활용한 python 및 numpy, numba 실행속도 측정

A JIT for Numeric Python [5/10]

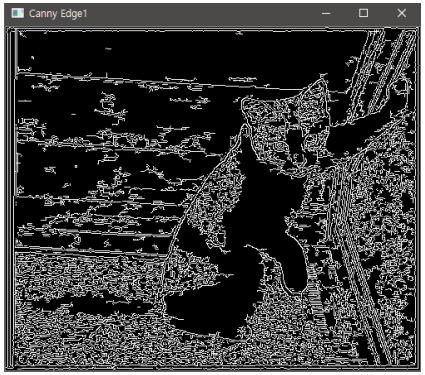
- **Numba Test (3/5)**

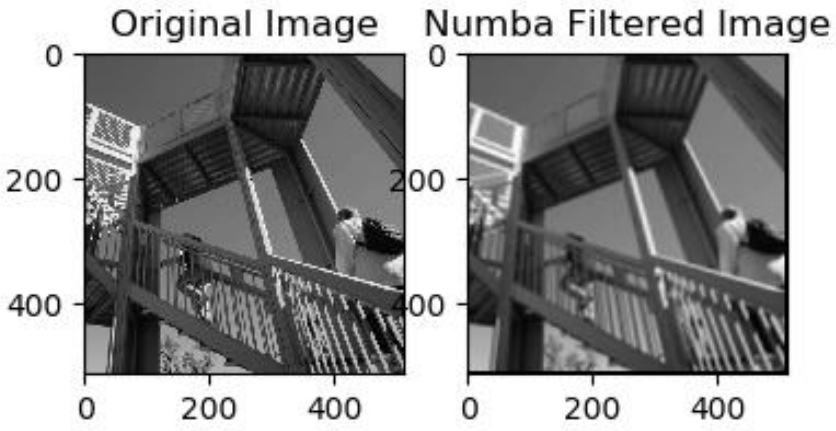
General(Matric Size: 10000x10000)				Binary File(Matric Size: 10000x10000)			
	Python	Numpy	Numba		Python	Numpy	Numba
Avg	**40.00695999**	**1.84720002**	**0.17242001**	**Avg**	**39.181740022**	**1.809740009**	**0.140719981**
1	39.39899993	1.76199985	0.17600012	1	39.671000004	1.786999941	0.141000032
2	40.08099985	1.82500005	0.16400003	2	39.746000052	1.792999983	0.137999773
3	39.51600003	1.72500014	0.15999985	3	39.626000166	1.761999846	0.152999878
4	39.52600002	1.71799994	0.16799998	4	42.916000128	1.960000038	0.145999908
5	39.44099998	1.74300003	0.16300011	5	41.544999838	1.964999914	0.148000002
6	39.80699992	1.79700017	0.16899991	6	40.095999956	1.894999981	0.141000032
7	39.46200013	1.80299997	0.15899992	7	40.209000111	1.866000175	0.157000065
8	39.40899992	1.77800012	0.18099999	8	39.541000128	1.821999788	0.149999857
9	39.44600010	1.77699995	0.16000009	9	40.074000120	1.796999931	0.141999960
10	39.42599988	1.77200007	0.17300010	10	39.546000004	1.773000002	0.137000084
11	40.01300001	1.88400006	0.16199994	11	39.331000090	1.736999989	0.139999866
12	40.07400012	1.81900001	0.16100001	12	39.368000031	1.835000038	0.134999990
13	41.54900002	1.93499994	0.23699999	13	39.549000025	1.878999949	0.138000011
14	42.93599987	1.92300010	0.23000002	14	39.875999928	1.838000059	0.146000147
15	41.84599996	1.87599993	0.18199992	15	40.017000198	1.783999920	0.138999939
16	40.58800006	1.91399980	0.17100000	16	39.624000073	1.815000057	0.148000002
17	40.54200006	1.85999990	0.17599988	17	39.967999935	1.788000107	0.138000011
18	38.99599981	1.75900006	0.15400004	18	39.321000099	1.827999830	0.134999990
19	39.86500001	1.81299996	0.16900015	19	39.320999861	1.820000172	0.148000002
20	40.30800009	1.87699986	0.16700006	20	39.236000061	1.818000078	0.148999929
⋮				⋮			
50				50			

Table. 2. jit decorator를 활용한 python 및 numpy, numba 실행속도 비교

A JIT for Numeric Python [6/10]

- **Numba Test (4/5)**






```
================================================================
Canny Edge Processing time : 0.08501911163330078 seconds
================================================================
================================================================
Numba Canny Edge Processing time : 0.0070018768310546875 seconds
================================================================
```

Fig. 6. Canny edge detection을 이용한 이미지 처리속도 측정

```
Numba Image Filter time = 0.012851

General Image Filter time = 10.444350
```

Fig. 7. Blur 효과를 이용한 이미지 처리속도 측정

A JIT for Numeric Python [7/10]

- **Numba Test (5/5)**

```
Step 1: Generator Loss: 0.725238, Discriminator Loss: 0.678302
Step 100: Generator Loss: 0.652346, Discriminator Loss: 0.589043
Step 200: Generator Loss: 3.693218, Discriminator Loss: 0.491116
Step 300: Generator Loss: 1.518299, Discriminator Loss: 0.389143
Step 400: Generator Loss: 2.266797, Discriminator Loss: 0.493464
Step 500: Generator Loss: 1.746747, Discriminator Loss: 0.349499
Step 600: Generator Loss: 2.271744, Discriminator Loss: 0.264137
Step 700: Generator Loss: 2.269647, Discriminator Loss: 0.445400
Step 800: Generator Loss: 2.580885, Discriminator Loss: 0.314098
Step 900: Generator Loss: 2.491915, Discriminator Loss: 0.242469
Step 1000: Generator Loss: 2.660532, Discriminator Loss: 0.222430
Numba Training time : 9.134794473648071 seconds
============================================================
Step 1: Generator Loss: 0.779860, Discriminator Loss: 0.666593
Step 100: Generator Loss: 1.224174, Discriminator Loss: 0.465956
Step 200: Generator Loss: 1.526144, Discriminator Loss: 0.367309
Step 300: Generator Loss: 2.033043, Discriminator Loss: 0.214565
Step 400: Generator Loss: 2.041603, Discriminator Loss: 0.230415
Step 500: Generator Loss: 2.226197, Discriminator Loss: 0.251711
Step 600: Generator Loss: 2.367590, Discriminator Loss: 0.370407
Step 700: Generator Loss: 2.569455, Discriminator Loss: 0.235937
Step 800: Generator Loss: 1.403335, Discriminator Loss: 0.384402
Step 900: Generator Loss: 3.798131, Discriminator Loss: 0.085129
Step 1000: Generator Loss: 4.261975, Discriminator Loss: 0.105004
General Training time : 5.907337188720703 seconds
============================================================
```

Fig. 8. DCGAN을 이용한 손글씨 이미지 학습 속도 측정

```
1/1 [==============================] - 0s 201ms/step - loss: 0.8696
Epoch 998/1000

1/1 [==============================] - 0s 204ms/step - loss: 0.9603
Epoch 999/1000

1/1 [==============================] - 0s 202ms/step - loss: 1.0621
Epoch 1000/1000

1/1 [==============================] - 0s 205ms/step - loss: 0.7123
============================================================
General Learning time : 230.65044260025024 seconds
============================================================
1/1 [==============================] - 0s 207ms/step - loss: 1.0213
Epoch 998/1000

1/1 [==============================] - 0s 199ms/step - loss: 0.6916
Epoch 999/1000

1/1 [==============================] - 0s 209ms/step - loss: 1.1388
Epoch 1000/1000

1/1 [==============================] - 0s 201ms/step - loss: 0.6531
============================================================
Numba Learning time : 442.20036935806274 seconds
============================================================
```

Fig. 9. ResNet50을 이용한 개/고양이 이미지 학습 속도 측정

A JIT for Numeric Python [8/10]

- **Numba Process (1/3)**

  1. Cpython 인터프리터는 jit decorator가 포함된 함수를 호출하여 구문을 분석하고 바이트코드(Bytecode)로 변환

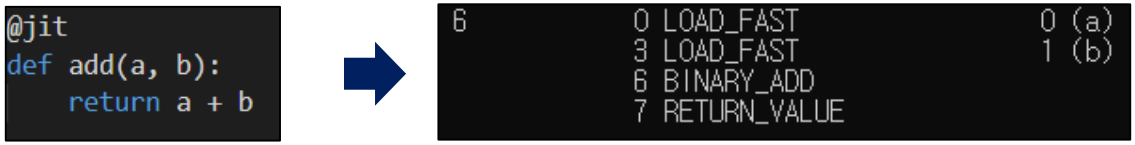


Fig. 10. Python 소스코드를 이용한 바이트코드 변환

  - LOAD_FAST: Local 변수에 대한 참조를 스택으로 push
  - BINARY_ADD: 스택에 최상단과 두번째 최상위 스택 항목을 더하여 스택의 최상단으로 (TOS = TOS1 + TOS2)
  - RETURN_VALUE: 함수의 caller에게 스택의 최상단 값 리턴

  2. 함수의 바이트코드를 분석하여 제어흐름그래프(Control Flow Graph)를 생성하고 제어흐름분석(Data Flow Analysis)을 통하여 cpython 인터프리터의 스택에 값이 어떻게 push/pop 되는지를 추적

A JIT for Numeric Python [9/10]

- **Numba Process (2/3)**

  3. 제어 흐름 및 데이터 분석이 완료되면 함수의 바이트코드를 스택 기반 가상 머신의 representation에서 레지스터 기반 representation인 Numba IR로 변환

  4. Numba IR에 적용된 local 변수 타입을 통하여 파라미터의 타입을 추론하며, 각 변수들에게 타입을 전파하기 위해 의존성 그래프(Dependency Graph)를 이용함
     만약 변수들에 대한 타입 추론이 실패할 경우, 모든 변수에 python object 타입이 할당되어 nopython 모드로 컴파일하지 못함

```
# --- LINE 6 ---

def add(a, b):

    # --- LINE 7 ---
    #   a = arg(0, name=a)  :: int64
    #   b = arg(1, name=b)  :: int64
    #   $0.3 = a + b  :: int64
    #   del b
    #   del a
    #   $0.4 = cast(value=$0.3)  :: int64
    #   del $0.3
    #   return $0.4
```

Fig. 11. 바이트코드를 이용한 Numba IR 변환

A JIT for Numeric Python [10/10]

- **Numba Process (3/3)**

  5. 타입 추론 후, Numba IR의 최적화를 위해 함수 내에 포함된 loop 및 array expression을 탐지하고 추출하여 새로운 함수로 Numba IR을 재작성
     (python operator가 ndarray 객체에 적용될 때만 expression 최적화 수행)

```
define i32 @"_ZN8__main__7add$241Exx"(i64* noalias nocapture %"retptr",
 {i8*, i32}** noalias nocapture %"excinfo", i64 %"arg.a", i64 %"arg.b")

{
entry:
  %"a" = alloca i64
  store i64 0, i64* %"a"
  %"b" = alloca i64
  store i64 0, i64* %"b"
  %"$0.3" = alloca i64
  store i64 0, i64* %"$0.3"
  %"$0.4" = alloca i64
  store i64 0, i64* %"$0.4"
  br label %"B0"
B0:
  %".7" = load i64, i64* %"a"
  store i64 %"arg.a", i64* %"a"
  %".10" = load i64, i64* %"b"
  store i64 %"arg.b", i64* %"b"
  %".12" = load i64, i64* %"a"
  %".13" = load i64, i64* %"b"
  %".14" = add nsw i64 %".12", %".13"
  %".16" = load i64, i64* %"$0.3"
  store i64 %".14", i64* %"$0.3"
  %".18" = load i64, i64* %"b"
  store i64 0, i64* %"b"
  %".20" = load i64, i64* %"a"
  store i64 0, i64* %"a"
  %".22" = load i64, i64* %"$0.3"
  %".24" = load i64, i64* %"$0.4"
  store i64 %".22", i64* %"$0.4"
  %".26" = load i64, i64* %"$0.3"
  store i64 0, i64* %"$0.3"
  %".28" = load i64, i64* %"$0.4"
  store i64 %".28", i64* %"retptr"
  ret i32 0
}
```

Fig. 12. LLVM IR

  6. 최적화된 Numba IR을 LLVM IR로 변환하고, LLVM IR은 LLVM JIT 컴파일러에 의해 컴파일 되어 실행되게 됨

Conclusion

- 본 논문에서는 python 환경에서 numpy 배열 및 루프의 반복을 이용하여 수치 계산을 수행할 때 발생하는 실행 속도 저하 문제를 개선하기 위한 LLVM 기반 JIT 컴파일러인 “Numba” 를 소개함