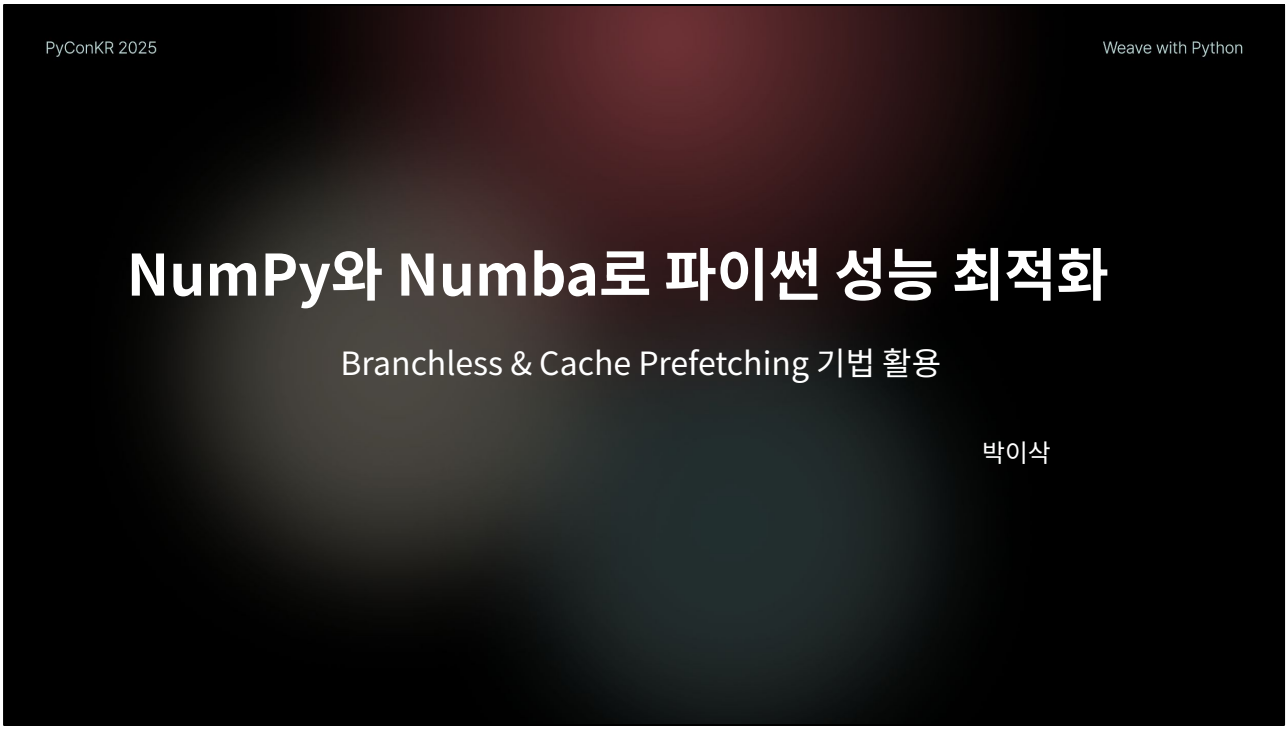
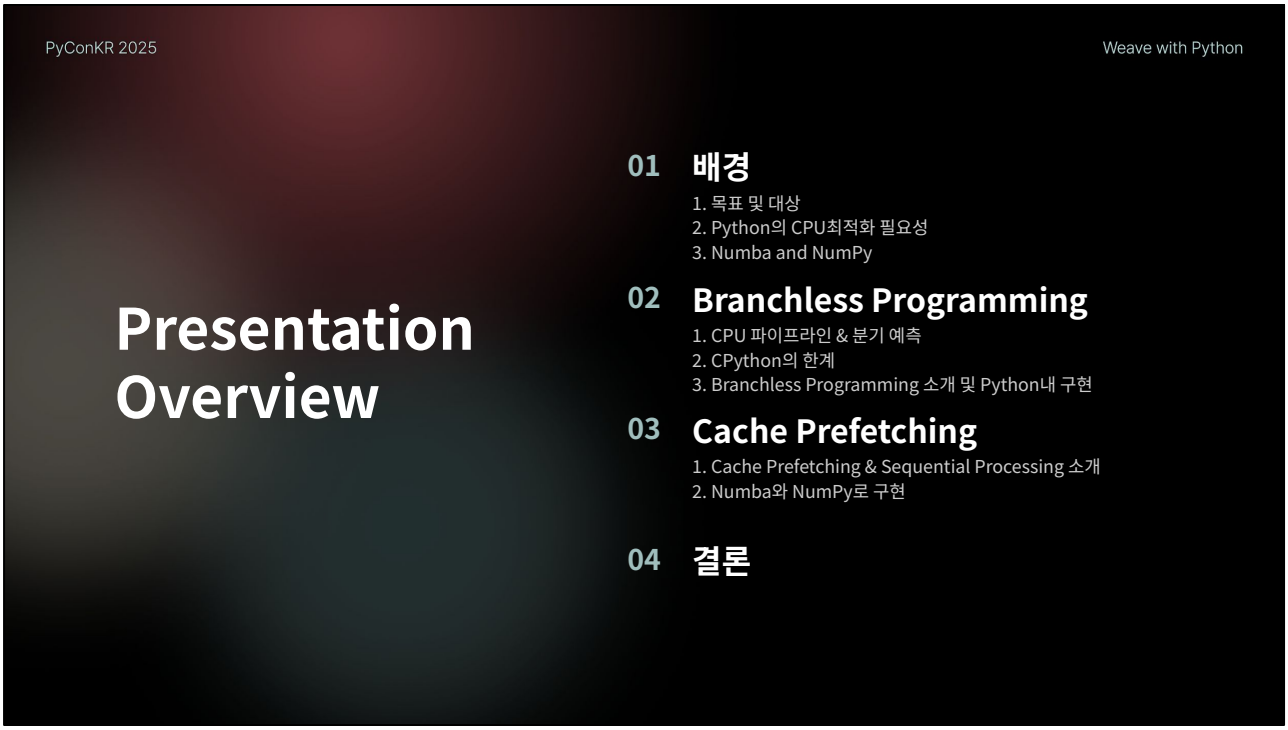


안녕하세요, 오늘 numpy와 numba로 python 최적화하는 방법 대해 발표하는 개발자 박이삭입니다. 반갑습니다.



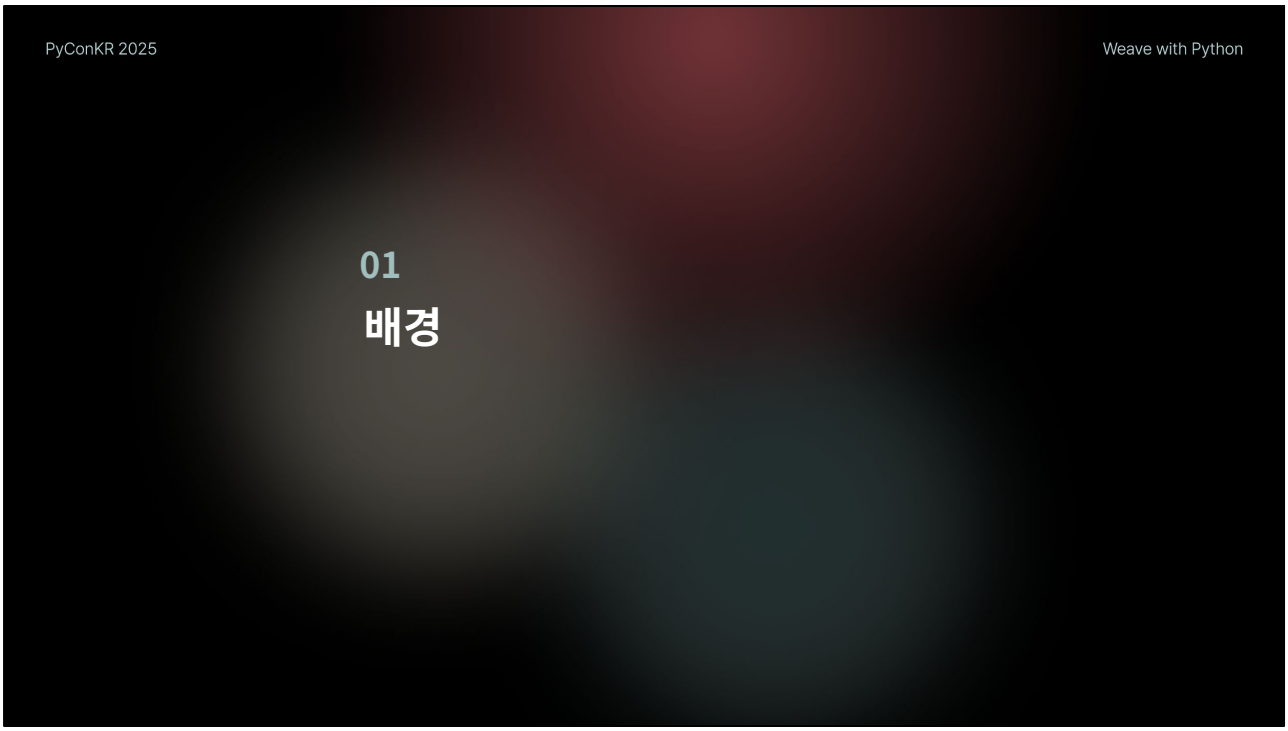
목차입니다. 처음 발표하려는 내용과 타겟 및 간단한 배경을 설명드리고, 상세설명에 들어가겠습니다.

자기소개

박이삭

소속: (주)휴이노
블로그: https://blog.i544c.com
트위터: https://x.com/i544c_park

저는 인공지능 기반의 디지털헬스케어 스타트업 휴이노에서 백엔드 Team Lead를 맡고있는 개발자 박이삭이라고 합니다.



먼저 오늘 발표에 대한 배경을 소개드리려고 합니다.

배경

목표 및 대상

목표

Python으로 CPU-bound 워크로드를 C/C++ 수준의 성능으로 최적화할 수 있음을 확인

Branchless Programming과 Cache Prefetching 기법을 통해 Python 코드의 CPU 효율성을 극대화하는 방법을 탐구

타겟

Python을 주로 사용하지만 성능 최적화에 관심 있는 개발자들 (e.g., 데이터 사이언티스트, ML 엔지니어)

CPU 파이프라인, 캐시 메모리 등에 대한 사전 지식이 있으면 좋지만, 발표에서 상세히 설명

이 발표의 목표는 Python으로도 일부 CPU-bound한 워크로드에서 C나 C++에 버금가는 성능을 낼 수 있다는 점을 보여드리는 것입니다. 보통 Python은 느리다고 생각되지만, 적절한 기법으로 이를 극복할 수 있습니다.

관심 있으실 분들로는 Python을 많이 사용하시고 성능 최적화에 관심 있는 분들을 생각했습니다. 예를 들어, 데이터 처리나 ML 모델을 다루는 분들입니다. CPU 파이프라인이나 캐시 같은 저수준 지식이 있으면 더 좋겠지만, 없어도 괜찮습니다. 발표 내내 기본 개념부터 상세히 설명하겠습니다. Branchless Programming과 Cache Prefetching 내용과 실전 예제를 중심으로 진행하려고 합니다.

배경

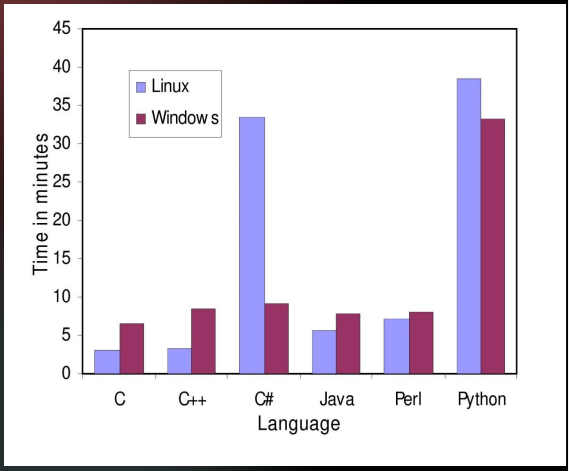
Python의 CPU최적화 필요성

Python은 C보다 최소 8배 이상 느림(인터프리터 오버헤드 때문)

IO-bound 워크로드 (e.g., database, network, disk I/O)에서는 큰 차이 없음

CPU-bound 워크로드에서 차이가 큼
시그널/이미지 처리, 커스텀 ML 모델, 대량 vectorized 데이터 처리 (통계, 선형 대수 등)

NumPy/Scikit-learn 제공 기능 외 **커스텀 로직 구현 시** 최적화 필수



Speed comparison of the BLAST parsing program

Source : https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-82/figures/3

Python의 성능 한계를 인정하면서도, 왜 CPU 최적화가 필요한지 설명하겠습니다. 잘 알려진 대로 Python은 C보다 최소 8배 느립니다. 이런 차이는 인터프리터의 오버헤드 등 에서 발생합니다. 오른쪽 차트는 BLAST로 여러 언어로 성능비교 테스트를 한 결과입니다. BLAST는 생물정보학에서 DNA, RNA, 단백질 서열을 비교하는 고성능 도구로, 주로 C로 구현되어 있습니다. 오른쪽 차트에서 보듯, BLAST의 C 구현은 Python 구현에 비해 최대 20배 이상 빠른 실행 속도를 보여줍니다. 하지만 Python는 모든 워크로드에서 똑같이 느린 건 아닙니다. 데이터베이스나 네트워크 같은 IO-bound 작업에서는 큰 차이가 안 보입니다.

즉 문제는 CPU-bound 워크로드입니다. 예를 들어, 시그널/이미지 프로세싱, 커스텀 ML 모델, 또는 대량의 벡터화된 데이터 처리처럼 CPU 시간이 많이 소모되는 경우죠. 여기서 Python의 약점이 드러납니다. NumPy나 Scikit-learn 같은 라이브러리가 제공하는 기본 기능으로는 괜찮지만, 커스텀 로직을 구현할 때 성능이 급락합니다. 분기 예측 실패나 캐시 미스로 인해 발생하는 지연이 그 원인중에 하나입니다. 오늘은 이런 문제를 Branchless와 Prefetching으로 어떻게 해결할지 보여드리려고 합니다.

배경

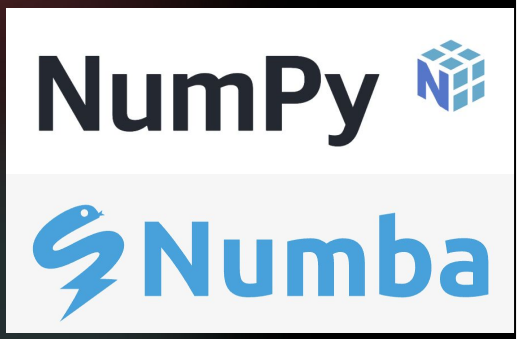
Numba and NumPy

NumPy

Python의 고성능 배열 연산 라이브러리
C로 구현되어 빠른 벡터화 연산과 메모리 연속성을 제공
대용량 데이터 처리와 행렬 연산에 최적화

Numba

Python 코드를 JIT 컴파일, 네이티브 머신 코드로 변환 라이브러리
루프 최적화와 SIMD 활용으로 CPU-bound 작업의 성능을 C/C++ 수준으로 끌어올림



Source :
https://numpy.org/
https://numba.pydata.org/

이제 Python에서 이런 최적화를 실현할 도구로 Numba와 NumPy를 소개하겠습니다. 먼저 NumPy는 배열 기반 연산을 위한 필수 라이브러리입니다. 내부적으로 C로 구현되어 있어서 벡터화된 연산이 매우 빠릅니다. Python 내 연속적인 배열을 관리할 수 있는 가장 좋은 방법입니다. 기본적으로 multi-threading, SIMD 등 백터연산의 최적화 되어있어 큰 어려움 없이 큰 데이터의 연산을 빠르게 할 수 있습니다.

Numba는 Python 코드를 런타임에 네이티브 코드로 컴파일해주는 JIT 컴파일러입니다. 데코레이터 하나로 루프를 최적화하고, SIMD 명령어를 자동으로 활용할 수 있습니다. numpy같은 배열 데이터 뿐만 아니라, list, dict 등 종류 데이터도 성능 부스트를 얻을 수 있습니다.

02

Branchless Programming

PyConKR 2025

Weave with Python

Branchless Programming

1. CPU 파이프라인 & 분기 예측

CPU 파이프라인

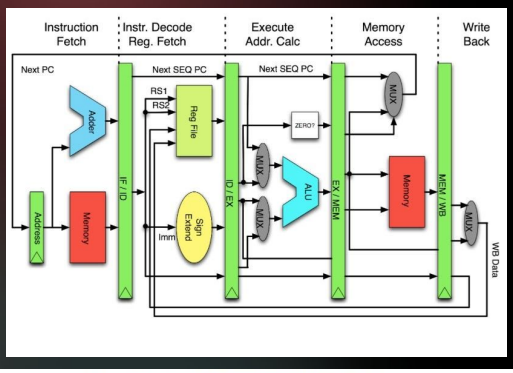
CPU는 명령어를 단계별로 처리

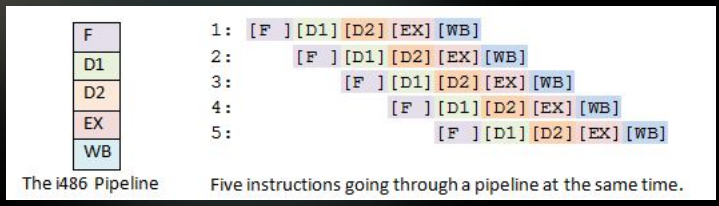
주요 단계: Fetch, Decode, Execute

파이프라인: 여러 명령어 동시 처리

성능 향상, 효율적 자원 사용

Source :
https://commons.wikimedia.org/wiki/File:Pipeline_MIPS.png
https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/a-journey-through-the-cpu-pipeline-r3115/





The i486 Pipeline

Five instructions going through a pipeline at the same time.

Branchless를 이해하기 먼저 CPU 파이프라인에 대해 알아야 됩니다. CPU의 branch 즉 분기에 대해 이해가 필요하기 때문입니다. CPU는 명령어를 단계별로 처리합니다. 주요 단계는 명령어를 가져오는 'Fetch'', 그 의미를 해석하는 'Decode' 그리고 실제로 실행하는 'Execute''입니다. 파이프라인은 이 단계들을 동시에 여러 명령어에 적용해 처리 속도를 높이는 기술입니다. 예를 들어, 한 명령어가 실행 중일 때 다음 명령어를 미리 가져옵니다. 이렇게 하면 CPU 자원을 효율적으로 사용해 전체 성능이 향상됩니다. 이미지로 보실 수 있듯이 동시에 여러 명령어를 순차적으로 실행하며 어느 단계가 되면 동시에 5개 모든 파이프라인을 실행하며 5배 효능성을 이루는 것을 보실 수 있습니다.

PyConKR 2025

Weave with Python

Branchless Programming

1. CPU 파이프라인 & 분기 예측

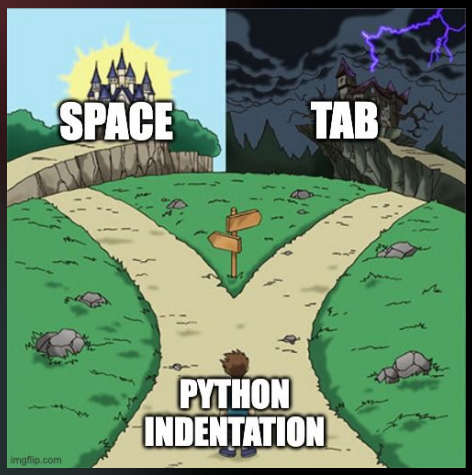
분기 예측

조건문, 반복문 등에서 코드 흐름 예측

CPU가 다음 명령어 미리 가져옴

예측 성공: 빠른 실행

예측 실패: 파이프라인 정지(Pipeline flush)



Source : https://imgflip.com/i/a39w0p

다음으로 분기 예측에 대해 알아보겠습니다. 프로그램에는 조건문이나 반복문처럼 코드 흐름이 갈라지는 부분이 많습니다. CPU는 이런 분기를 미리 예측해 다음 명령어를 가져옵니다. 예측이 맞으면 실행이 빨라지지만, 틀리면 파이프라인이 멈춰버립니다. 이게 왜 중요한지 예상해 볼 수 있습니다. 파이프라인이 멈춰버리면 다음 단계 명령어는 flush됩니다. 그로 예측 성공률이 높을수록 프로그램이 더 빠르게 동작합니다.

PyConKR 2025

Weave with Python

Branchless Programming

1. CPU 파이프라인 & 분기 예측

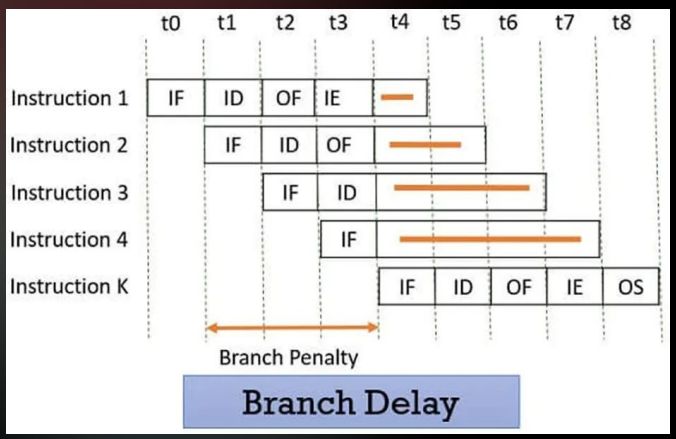
분기 실패 패널티

예측 틀리면 파이프라인 비움

새 명령어 다시 가져옴

시간과 자원 낭비

성능 저하, 특히 깊은 파이프라인



Source :
https://levelup.gitconnected.com/how-does-the-processor-can-predict-the-next-instruction-to-execute-a28336edaeba

분기 예측이 실패했을 때의 패널티를 설명 드리겠습니다. 예측이 틀리면 CPU는 잘못된 명령어들을 파이프라인에서 모두 비우고, 올바른 명령어를 다시 가져와야 합니다. 이 과정에서 시간과 자원이 많이 낭비됩니다. 특히 현대 CPU처럼 파이프라인이 깊을수록 이 패널티가 커집니다. 결과적으로 프로그램 성능이 크게 떨어질 수 있습니다. 이미지로 보실 수 있듯이 주황색의 패널티 시간 구간동안은 CPU는 놀게 되면서 동시간 효율성이 떨어지게 됩니다.

여기까지 CPU와 분기에 대해 설명을 드렸습니다. 여기서 branchless programming을 하려면 극복해야 하는 Python의 한계에 대해 설명드리도록 하겠습니다. CPython은 Python의 표준 구현체로, 소스 코드를 바이트코드로 변환한 뒤 CPython 가상 머신에서 실행합니다. 이 과정에서 바이트코드를 디코딩하며 많은 분기가 발생해 원래 실행 코드 외에 추가적인 오버헤드가 생깁니다. 가장 오른쪽 코드는 CPython의 bytecode를 디코딩하는 로직을 수도코드로 설명하고 있습니다. 수 많은 opcode를 여러 분기로 검사하고 실행하는 것을 확인 할 수 있습니다. 런타임 뿐만 아니라 Python의 동적 타입 언어 특성상 리스트와 같은 객체에 다양한 타입이 포함될 수 있어, 매 루프마다 타입 검사를 수행하는 오버헤드가 발생합니다. 왼쪽 이미지는 파이썬 리스트 내 여러 타입이 있을 수 있는 언어의 특징을 보여줍니다. 여기에 가비지 컬렉션(GC)과 전역 인터프리터 락(GIL)도 성능에 영향을 미치는 요소입니다.

PyConKR 2025

Weave with Python

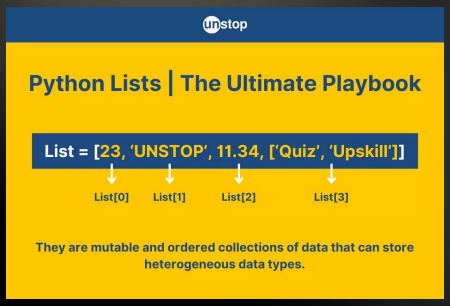
Branchless Programming

2. CPython의 한계

최적화 방향

CPython VM(Bytecode decode)

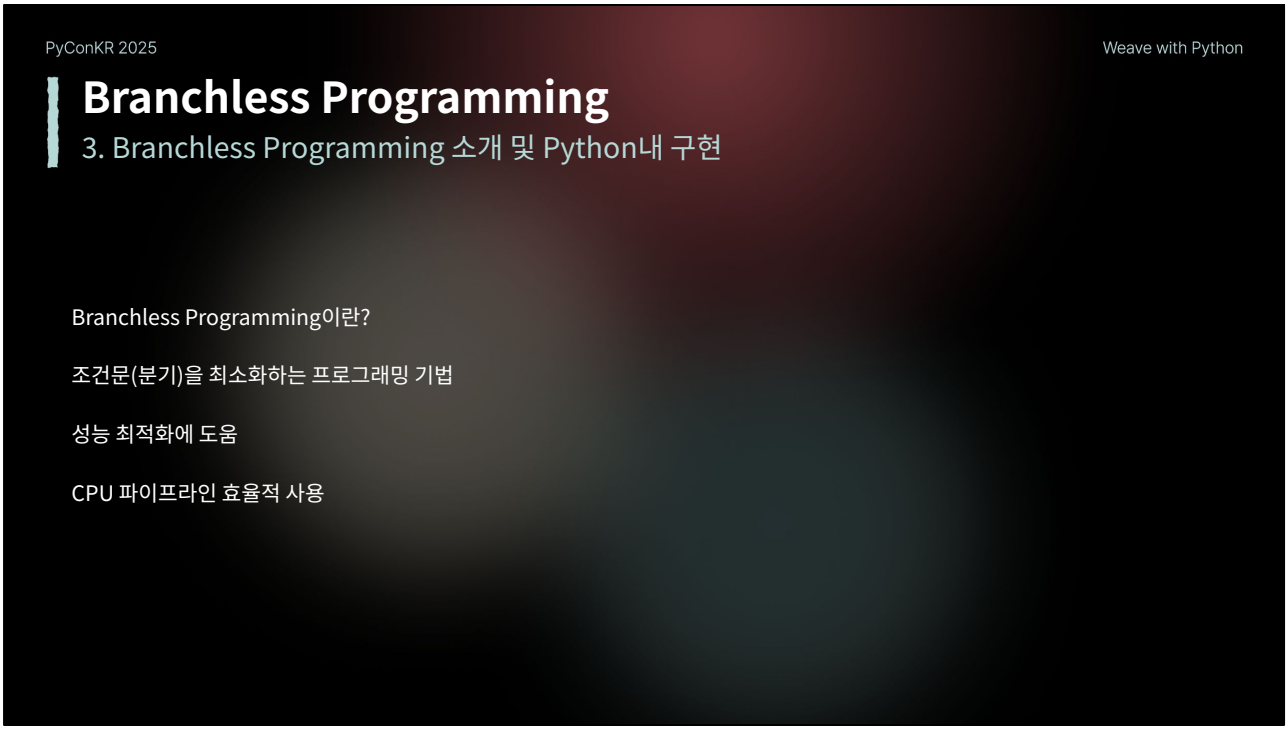
동적 타입




```c
for (;;) {
    opcode = NEXTOP();
    switch (opcode) {
        case TARGET(NOP): {
            DISPATCH();
        }
        case TARGET(LOAD_CONST): {
            PyObject *value = GETITEM(consts, NEXTARG());
            PUSH(value);
            DISPATCH();
        }
        case TARGET(STORE_NAME): {
            PyObject *name = GETITEM(names, NEXTARG());
            PyObject *v = POP();
            int err = PyObject_SetAttr(f->f_globals, name, v);
            Py_DECREF(v);
            if (err != 0) goto error;
            DISPATCH();
        }
        case TARGET(POP_TOP): {
            PyObject *value = POP();
            Py_DECREF(value);
            DISPATCH();
        }
        /* ... (other opcodes like BINARY_ADD, CALL_FUNCTION, etc.) */
        default:
            PyErr_Format(PyExc_SystemError,
                         "unknown opcode: %d", opcode);
            goto error;
    }
}
```

Source :
https://github.com/python/cpython/blob/main/Python/ceval.c
https://unstop.com/blog/python-list

저는 이 중 CPython의 runtime과 동적타입의 overhead를 극복하며 파이썬에서 branchless programming을 구현해보려고 합니다.



Branchless Programming의 기본 개념을 소개합니다. Branchless programming은 if-else나 switch 같은 조건문, 즉 분기를 최소화하거나 제거하는 프로그래밍 기법입니다. 이전 까지 소개드린 CPU 파이프라인 flush를 최소화 하여 CPU리소스를 최적화 할 수 있습니다.

이제 branchless programming의 주요 기법을 살펴보겠습니다. 분기를 제거하기 위해 여러 가지 방법을 사용합니다. 여기서 설명에 사용되는 code들은 이해하시기 편하게 python으로 되있지만 수도코드로 생각하시면 좋을꺼 같습니다. 이전에 말씀 드린대로 python의 한계로 인해 이 코드들은 실제 branchless programming이 되지 못합니다.

첫째로 비트연산 방법이 있습니다. 비트 연산은 조건문을 논리 연산으로 대체해 분기를 없앱니다. 예시로 절댓값 계산하는 함수를 branchless로 바꿀 때 64bit int 의 sign 값을 mask로 negate화 하는 방식을 활용한 기법입니다. 분기를 사용하는 위 코드는 if와 else를 list comprehension 내서 사용하게 되며 분기를 발생시킵니다. 분기없는 코드는 산수 operation, bit operation으로 분기를 대체했습니다.

다음은 룩업 테이블입니다. 룩업 테이블은 미리 계산된 결과를 테이블에 저장해 조건 체크를 줄입니다. 예시로 4bit 룩업테이블로 페러티 계산의 최적화 방법을 보실 수 있습니다. 분기가 있는 코드는 매 8개 비트를 루프로 돌며 종료여부를 분기로 확인하고 있습니다. 그에 비해 분기없는 코드는 미리 계산 된 룩업테이블과 4개 추가 연산으로 분기를 없엔 것을 확인 하실 수 있습니다. Opcode 디코딩 이나 함수테이블 같은 수 많은 조건검사가 필요한 경우에도 룩업테이블은 많이 사용됩니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

분기 대신 사용하는 방법

비트 연산

룩업 테이블

산술 연산

```python
from typing import List, Tuple

# Before: Branched implementation to find the maximum of two integers
def branched_max(pairs: List[Tuple[int, int]]) -> List[int]:
    return [a if a >= b else b for a, b in pairs]

# After: Branchless implementation using arithmetic operations
# Uses arithmetic to compute max(a, b) without conditionals
def branchless_max(pairs: List[Tuple[int, int]]) -> List[int]:
    return [((a + b + abs(a - b)) // 2) for a, b in pairs]

# Example usage (practical for max operations in algorithms, e.g., sorting or comparisons)
pairs: List[Tuple[int, int]] = [(5, 3), (2, 7), (-1, 0), (4, 4)]
print("Branched:", branched_max(pairs))   # Output: [5, 7, 0, 4]
print("Branchless:", branchless_max(pairs)) # Output: [5, 7, 0, 4]
```

If $a \geq b$, then $|a-b| = a-b$, so: $\frac{a+b+(a-b)}{2} = \frac{a+b+a-b}{2} = \frac{2a}{2} = a$

If $b > a$, then $|a-b| = b-a$, so: $\frac{a+b+(b-a)}{2} = \frac{a+b+b-a}{2} = \frac{2b}{2} = b$

세번째로는 arithmetic 연산으로 branchless programming을 구현하는 방법입니다.
if문 하나 줄이려고 operation을 엄청 추가한 특징을 보실 수 있습니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

장점

성능 향상(일부케이스)

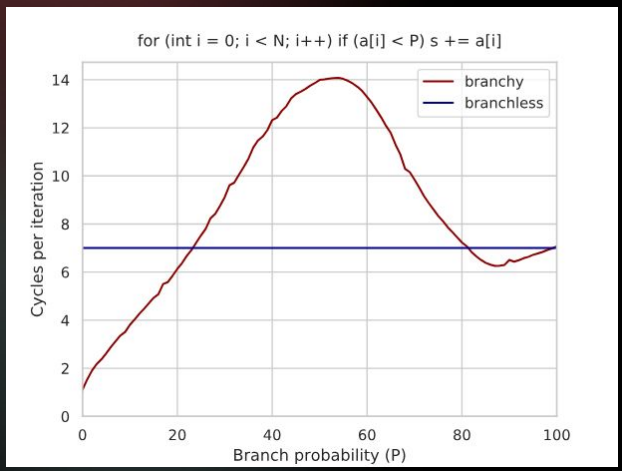
예측 가능한 실행 시간

단점

코드 가독성 저하

디버깅 어려움

일부 환경에서 비효율적



Source :
https://en.algorithmica.org/hpc/pipelining/branchless/

Branchless programming의 장단점을 소개해 드리겠습니다. Branchless programming의 장점으로는 분기 예측 실패를 줄여 일부 케이스에 성능을 향상시키고, 실행 시간을 예측 가능하게 만드는 점이 있습니다. 이는 실시간 시스템에서 특히 유용합니다. 반면, 단점으로는 비트 연산이나 복잡한 산술 연산으로 인해 코드 가독성이 떨어질 수 있습니다. 또한, 디버깅이 어려워지고, 일부 고수준 언어 에서는 성능 이점이 미미할 수 있습니다. 따라서 적용 시 주의가 필요합니다. 오른쪽 차트는 branchless 프로그램과 branched 기법의 성능차이를 보여줍니다. 입력값과 분기의 변수가 없기 때문에 분기없는 코드는 일정한 예측이 가능한 성능을 보여주고 있습니다. 설명드린 실시간 시스템에서 jitter나 spike가 발생하면 안되는 경우에 적절히 사용될 수 있습니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

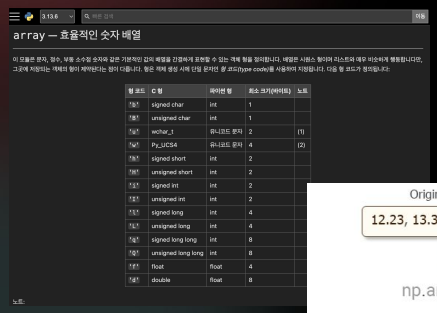
Pure Python과 Branchless Programming

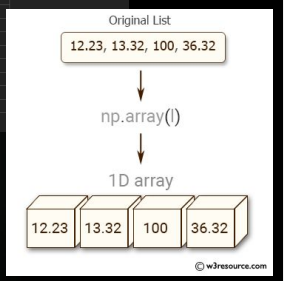
Pure Python: Branchless 불가능

이유: 동적 타입, 인터프리터 오버헤드, 조건문 최적화 불가

대안: Numba, 정적타입

효과: 컴파일 언어급 Branchless 구현





Source :
https://docs.python.org/ko/3.13/library/array.html
https://www.w3resource.com/python-exercises/numpy/python-numpy-exercise-2.php

Branchless Programming은 CPU 분기를 줄여 성능을 높이는 기법인데, 이 전에 말씀드린데로 순수 Python에서는 동적 타입, 인터프리터 오버헤드, 조건문 최적화 불가로 구현이 어렵습니다. 동적 타입은 런타임에 타입을 결정해 내부 분기를 유발하고, 인터프리터는 저수준 제어를 제한하죠. 하지만 Numba나 array 데이터 타입을 사용하면 컴파일 언어처럼 효율적인 Branchless 코드를 작성할 수 있습니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

Branchless Programming의 한계

모던 CPU의 평범 workload 분기예측율 95%

추가 연산으로 인한 overhead 발생

컴파일러의 최적화 고려

Source :
https://www.intel.com/content/www/us/en/products/details/processors/xeon.html

단 branchless programming은 만병통치약이 될 수 없습니다. 이미 모던 CPU에서는 평범한 workload에서 이미 95%의 분기예측율을 가지고 있습니다. 그래서 없이도 어느정도의 성능은 보장이 됩니다. 또한 branchless programming을 하며 추가 연산이 필요하게 되어 오히려 성능이 낮아지는 경우도 있습니다. 그래서 사용 전 benchmark이 꼭 필요합니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

적합한 경우

고성능 컴퓨팅

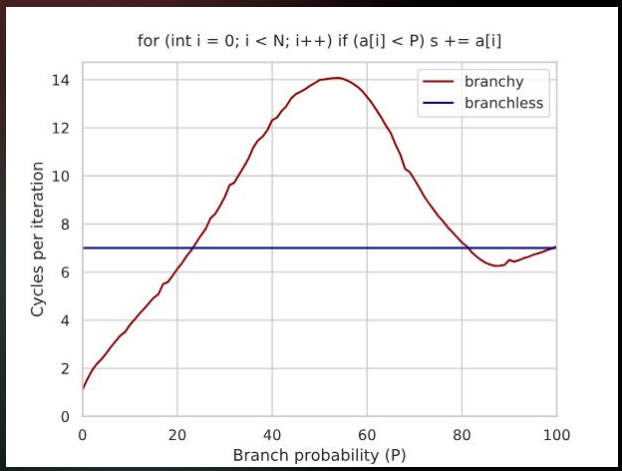
실시간 시스템

그래픽/게임 엔진

주의할 점

코드 복잡도 고려

벤치마킹 필수



Source : https://en.algorithmica.org/hpc/pipelining/branchless/

가장 많이 사용되는 환경은 고성능 컴퓨팅, 실시간 시스템, 그래픽 처리, 게임 엔진과 같이 성능이 매우 중요한 경우입니다. 말씀드렸듯이 branch prediction이 이미 잘 되고 있는 경우가 있을 수 있고 추가 연산이 오히려 성능을 저하 할 수 있습니다. 도한 복잡한 코드로 인해 유지보수가 어려워질 수 있습니다. 적용 전 반드시 벤치마킹을 통해 성능 차이를 확인해야 합니다. 또한, 컴파일러가 자동으로 분기를 최적화할 수 있는 경우도 있으므로, 이를 고려해야 합니다. 디스어셈블로 최적화 전 후를 비교할 필요도 있습니다.

PyConKR 2025

Weave with Python

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

예시 테스트 1

8 bit parity 계산하는 함수

Numba SIMD disable

Branched: 8 bit loop

Branchless: Look up Table 사용

```python
import numpy as np
from numba import jit
import numba.types as types

# Precomputed LUT for 4-bit chunks (16 entries for 0 to 15)
PARITY_LUT = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0], dtype=np.int64)

# Before: Branched implementation to compute parity of an 8-bit integer
@jit(nopython=True)
def branched_parity(numbers: np.ndarray) -> np.ndarray:
    result = np.empty(len(numbers), dtype=np.int64)
    for i in range(len(numbers)):
        count = 0
        n = numbers[i] & 0xFF  # Ensure 8-bit by masking
        while n:
            count += n & 1  # Check least significant bit
            n >>= 1  # Shift right
        result[i] = count % 2  # 0 for even, 1 for odd
    return result

# After: Branchless implementation using a lookup table and bit operations
@jit(nopython=True)
def branchless_parity(numbers: np.ndarray) -> np.ndarray:
    result = np.empty(len(numbers), dtype=np.int64)
    for i in range(len(numbers)):
        num = numbers[i]
        result[i] = PARITY_LUT[num & 0xF] ^ PARITY_LUT[(num >> 4) & 0xF]
    return result

# Example usage (practical for parity checks, e.g., in data transmission)
# 0: 00000000, 1: 00000001, 3: 00000011, 5: 00000101, 255: 11111111
numbers = np.array([0, 1, 3, 5, 255], dtype=np.int64)
print("Branched:", branched_parity(numbers))  # Output: [0 1 0 0 0] (all even parity)
print("Branchless:", branchless_parity(numbers))  # Output: [0 1 0 0 0] (all even parity)
```

이제 python으로 branchless programming을 구현해 보려고 합니다. 예시로 8 bit array의 bit count합이 홀수 짝수인지 계산하는 로직을 두 branch, branchles로 구현하여 비교해보려고 합니다. 앞서 말씀드렸듯이 python의 interpreter와 동적타입의 한계를 극복하기 위해서 numba의 jit과 numpy의 정적배열을 사용했습니다. popcount를 하는 방법이 여러게 있는것을 알지만 branchless를 쉽게 설명드리기 위한 예시라고 생각해주시면 좋을꺼 같습니다. Branched 방법은 모든 8bit를 loop 돌며 1 bit의 갯수를 하나씩 세는 방식입니다. Branchless 방법은 이미 계산 된 4 bit parity정보를 Look up table로 사용하여 두 4bit의 결과를 합하는 방식입니다. Nested loop가 없는것을 알 수 있습니다. 의미있는 비교를 위해서 numba의 SIMD컴파일 옵션을 끄고 테스트를 진행했습니다. Branchless programming은 쉽게 백터화 되기 때문에 분기예측의 패널티를 테스트하기위해서 동일 백터를 사용안하도록 했습니다.

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

Branched Parity Assembly

And, add, shift right 단순 동작

비교 후 jmp

```
n = numbers[i] & 0xFF
while n:
    count += n & 1
    n >>= 1
result[i] = count % 2
```

```
movzbl (%r8,%r11,8), %ebx  # Load low byte of int64 element
xorl %esi, %esi            # Init parity accumulator
testq %rbx, %rbx           # If zero?
je .LBB0_12                # Yes, store 0 (even)
movq %rbx, %rdi            # Copy for shift
.p2align 4, 0x90
.LBB0_10:
    movl %ebx, %ecx        # ecx = current value
    andl $1, %ecx          # LSB
    addq %rcx, %rsi        # Accumulate
    shrq %rdi              # Shift right
    cmpq $2, %rbx          # While original >1 (but updates rbx=rdi)
    movq %rdi, %rbx
    jae .LBB0_10           # Loop
andl $1, %esi              # Parity = accum %2
.LBB0_12:
    movq %rsi, (%r10,%r11,8)  # Store
```

Branched 방법의 머신코드입니다. 그 중 가장 특징되는 8 bit를 loop도며 각 bit를 검사하는 부분입니다.

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

Branchless Parity Assembly

조건부 jmp가 없음

```
num = numbers[i]
result[i] = PARITY_LUT[num & 0xF] ^ PARITY_LUT[(num >> 4) & 0xF]
```

```
movq (%r10,%rdx,8), %rbx    # Load full int64
movl %ebx, %edi             # Low 32
andl $15, %edi              # Low nibble (bits 0-3)
shrq %rbx                   # Shift right 1 (full 64, but assumes high=0)
andl $120, %ebx             # (bits 4-7 >>1) & 0x78 = high nibble *8 effective
movq (%rbx,%rsi), %rbx      # High nibble parity lookup
xorq (%rsi,%rdi,8), %rbx    # XOR low nibble parity
movq %rbx, (%r11,%rdx,8)    # Store (0 or 1)
```

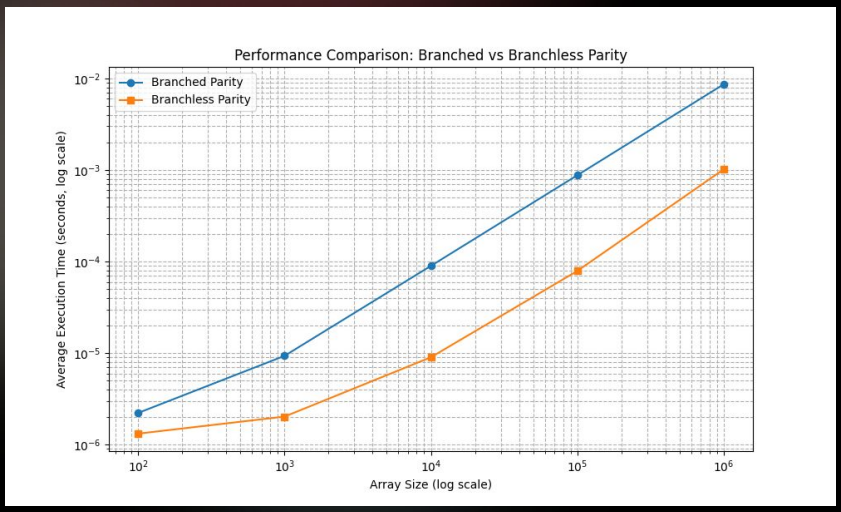
Branchless 방법의 가장 코어 어셈블리 코드입니다. numba의 llvm이 SIMD로 데이터 처리를 백터화 컴파일한 특징이 보입니다. branched코드에 비해 간단해져 컴파일러가 백터화를 할 수 있었습니다. 조건부. 가장 중요한 부분은 조건부 jump가 없습니다. 즉 branch가 없다는 것입니다.

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

Performance 비교

8 ~ 11배 성능 부스트



최대 11배 정도의 의미있는 성능 boost를 볼 수 있습니다.

Branchless Programming

3. Branchless Programming 소개 및 Python내 구현

예시 테스트 2

CTZ(Count Trailing Zeros) 계산 함수

Numba SIMD disable

Branched: 32 bit loop

Branchless: Look up Table 사용

```python
# Precomputed De Bruijn LUT (for 32-bit)
DEBRUIJN_LUT = np.array([
    0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
    31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
], dtype=np.int32)
MAGIC = 0x077CB531

# Branched implementation
@njit
def ctz_branched(numbers: np.ndarray) -> np.ndarray:
    result = np.empty(len(numbers), dtype=np.int32)
    for i in range(len(numbers)):
        x = numbers[i]
        if x == 0:
            result[i] = 32
            continue
        count = 0
        while (x & 1) == 0:
            count += 1
            x >>= 1
        result[i] = count
    return result

# Branchless implementation using De Bruijn sequence
@njit
def ctz_branchless(numbers: np.ndarray) -> np.ndarray:
    result = np.empty(len(numbers), dtype=np.int32)
    for i in range(len(numbers)):
        x = numbers[i]
        is_zero = int(x == 0)
        isolated = x & (0xFFFFFFFF - x + 1)  # x & -x, but safe for Python uint32 emulation
        hash_val = ((isolated * MAGIC) & 0xFFFFFFFF) >> 27
        ctz_val = DEBRUIJN_LUT[hash_val]
        result[i] = (1 - is_zero) * ctz_val + is_zero * 32
    return result
```

두 번 째 예시입니다. Count Trailing Zeros라는 연산입니다. integer의 Least Significant Bit 0이 몇개인지 계산하는 것인데요, 최적화 알고리즘, 수학연산, 압축, 암호 등에 사용된다고 합니다.

여기도 Numba의 SIMD 컴파일 옵션을 disable 했습니다.

Branchless Programming

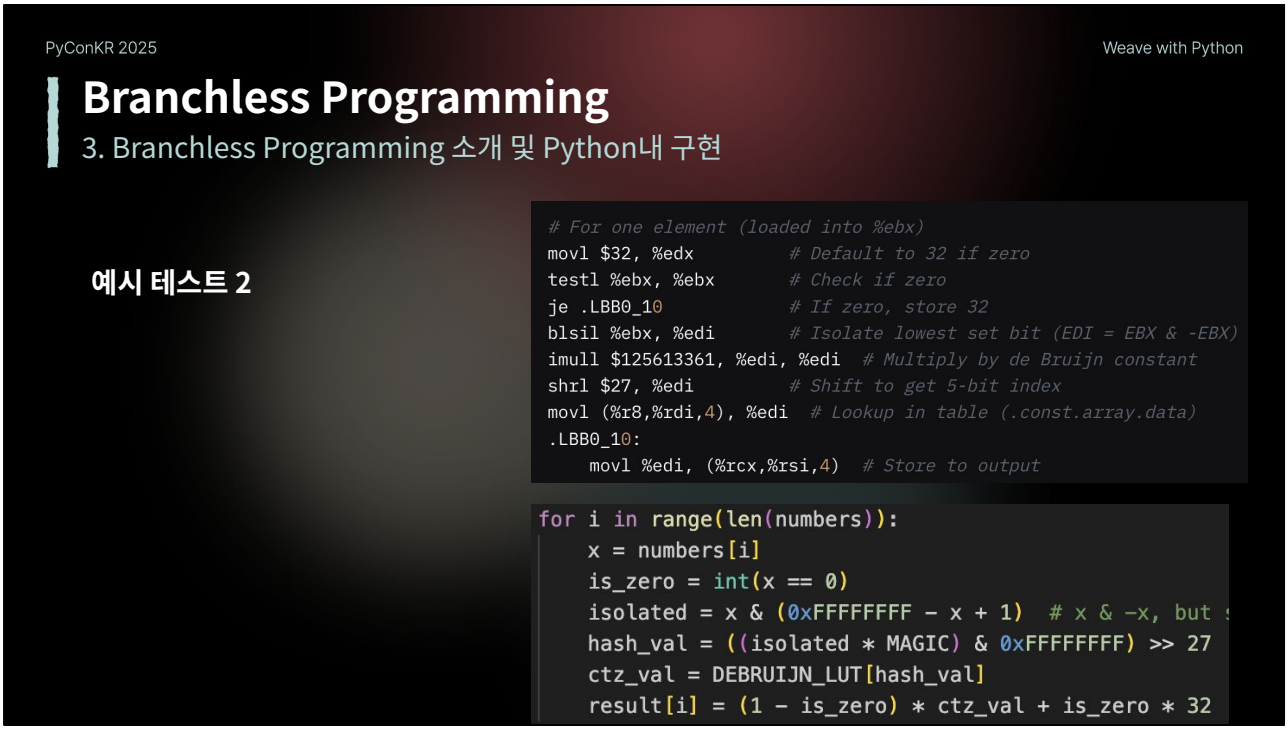
3. Branchless Programming 소개 및 Python내 구현

Branches CTZ Assembly

```
for i in range(len(numbers)):
    x = numbers[i]
    if x == 0:
        result[i] = 32
        continue
    count = 0
    while (x & 1) == 0:
        count += 1
        x >>= 1
    result[i] = count
```

```
# For one element in the pair (loaded into %ebx)
movl $32, %esi          # Default to 32 if zero
movl $32, %ecx
testq %rbx, %rbx        # Check if value is zero
je .LBB0_12             # If zero, store 32
xorl %ecx, %ecx         # Initialize count to 0
testb $1, %bl           # Check LSB
jne .LBB0_12            # If set, CTZ=0, store
xorl %ecx, %ecx         # Re-init count
movq %rbx, %rdi         # Copy value for shifting
.p2align 4, 0x90
.LBB0_11:               # Shift loop
    incl %ecx           # Increment count
    shrq %rdi           # Right shift
    testb $2, %bl       # Check if next bit was set (note: this s
    movq %rdi, %rbx
    je .LBB0_11         # Loop if not set
.LBB0_12:
    movl %ecx, (%r10,%r11,4)  # Store count to output
```

Branched 함수의 어셈블리입니다. 여기도 여러 비교와 점프가 있는 것을 확인 하실 수 있습니다.



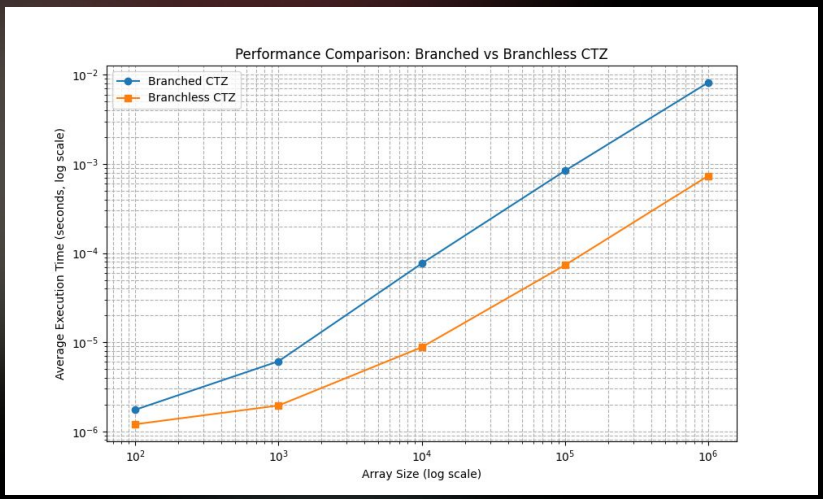
Branchless 어셈블리입니다. 분기가 최소화 된 것을 확인 할 수 있습니다.

Branchless Programming

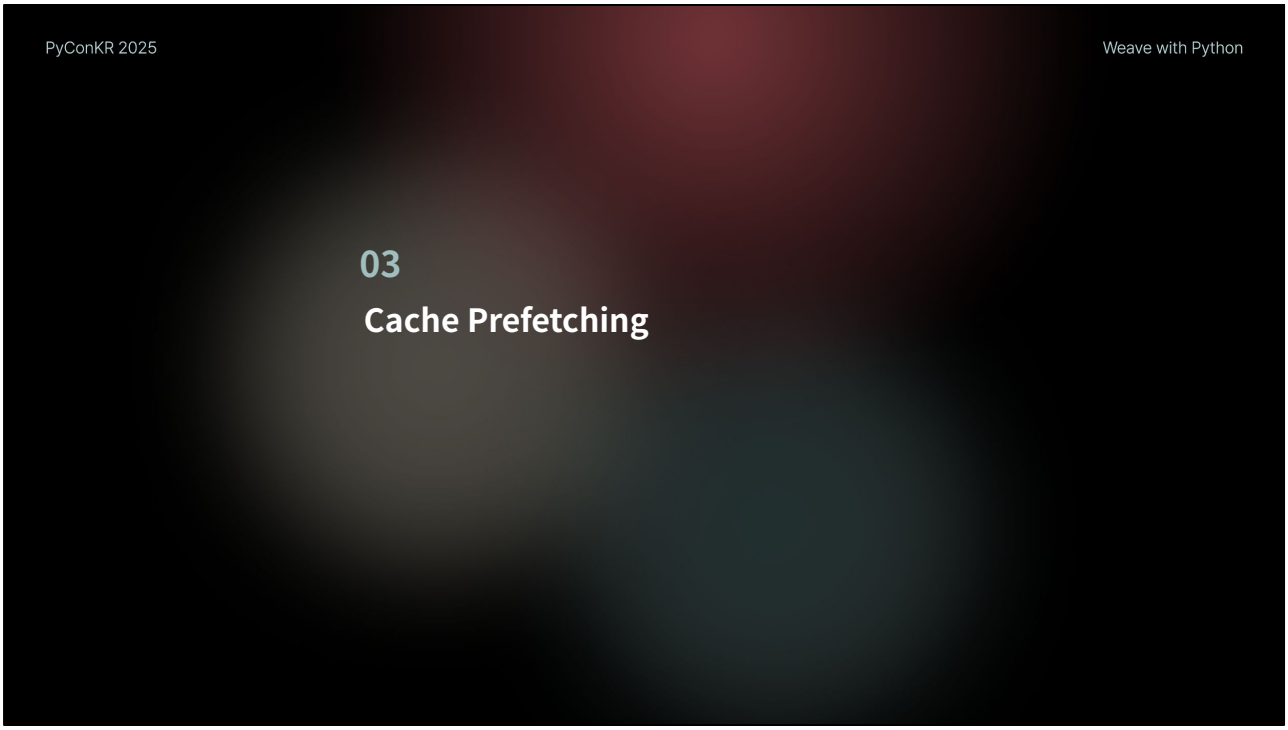
3. Branchless Programming 소개 및 Python내 구현

Performance 비교

8 ~ 11배 성능 부스트



8 ~ 11배 정도의 의미있는 성능 boost를 볼 수 있습니다.



다음은 cache prefetching에 대해 소개드리려고 합니다

PyConKR 2025

Weave with Python

Cache Prefetching

1. Cache Prefetching & Sequential Processing 소개

개요

CPU 캐시 성능의 중요성

CPU-메모리 속도 차이 병목

Computer Latency in Human Scale

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to AUS	183 ms	19 years
SCSI command time-out	30 s	3000 years

Source : https://x.com/aleksagram/status/541720043545849856?lang=bg

현대 컴퓨팅에서 CPU와 메모리 속도 차이가 큰 병목이 됩니다. 옆의 latency scale테이블을 보시면 메모리접근이 얼마나 L1 캐시보다 오래 걸리는 지 알 수 있습니다. 그래서 메모리의 병목을 줄이는 방법을 사람들은 고민했었습니다.

Cache Prefetching

1. Cache Prefetching & Sequential Processing 소개

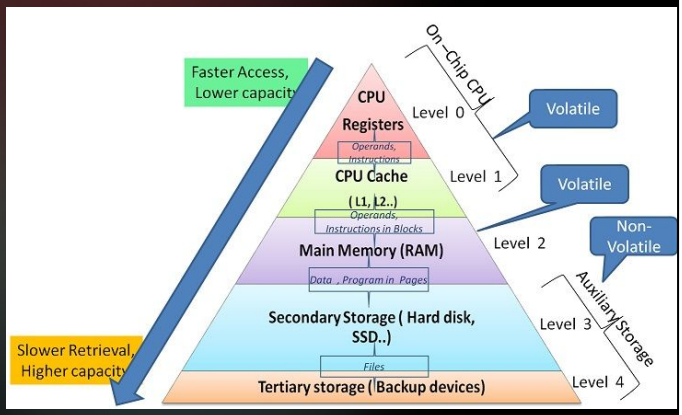
Cache 계층 구조

메모리 계층: 레지스터 > L1/L2/L3 캐시 > RAM > 저장소

캐시 역할: 자주 쓰는 데이터 저장

공간/시간 지역성 활용

캐시 미스: 미스 패널티로 지연 발생



Source : https://witscad.com/course/computer-architecture/chapter/memory-characteristics-and-organisation

캐시의 중요성을 말씀드리기 위해 캐시의 계층 구조를 설명드리려고 합니다. 메모리 계층은 레지스터가 가장 빠르고, 캐시 L1부터 L3, RAM, 저장소 순으로 느려집니다. 캐시는 자주 사용하는 데이터를 저장해 빠른 접근을 돕고, 공간 지역성과 시간 지역성을 활용합니다. 하지만 캐시 미스가 발생하면 주 메모리에서 데이터를 가져와 지연이 생깁니다.

Cache Prefetching

1. Cache Prefetching & Sequential Processing 소개

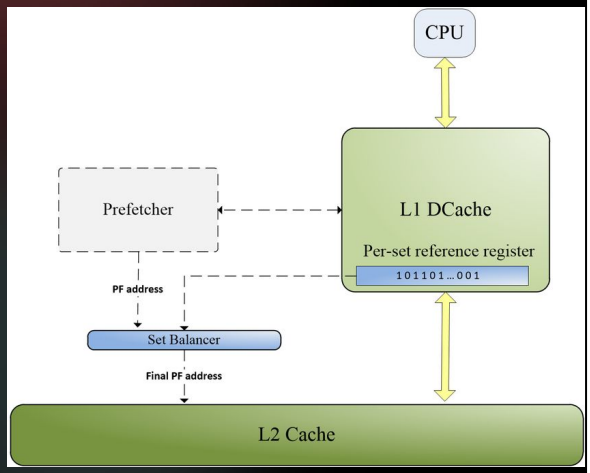
캐시 Prefetch 기술

정의: 미래 데이터 예측해 미리 로드

미스 줄여 지연 최소화

하드웨어: CPU 자동 예측

소프트웨어: 코드로 명시 (e.g. builtinprefetch)



Source :
https://www.researchgate.net/figure/Cache-Prefetcher-with-Set-Balancer-Prefetching_fig3_281905693

Cache prefetching이란 현재 사용 중인 cache의 주소로 미래 사용할 메모리 데이터를 미리 캐시로 가져와 cache 미스를 줄이는 기법입니다.캐시 미스는 최악 케이스로 RAM까지 데이터를 받아야 됩니다. 이는 약 350 CPU tick만큼 시간이 소요되고 CPU입장에서는 엄청 오래 기다리게 되어 성능에 영향을 주게 됩니다. Cache prefetch는 HW와 SW가 있습니다. 하드웨어 prefetch는 CPU가 자동으로 하고, 소프트웨어는 프로그래머가 코드로 지정합니다.

PyConKR 2025

Weave with Python

Cache Prefetching

1. Cache Prefetching & Sequential Processing 소개

Sequential Processing 및 이점

정의: 메모리 순서대로 데이터 처리

이점: 공간 지역성 활용, 히트율 ↑

RAM random access 문제: 미스 증가, 지연 ↑

예: 배열 순차 접근 vs 랜덤 (Python/Numpy)

Computer Latency in Human Scale

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to AUS	183 ms	19 years
SCSI command time-out	30 s	3000 years

Source : https://x.com/aleksagram/status/541720043545849856?lang=bg

Sequential processing에 대해 그리고 그 장점에 대해 설명드리겠습니다. Sequential processing이란 말 그대로 순차적으로 데이터를 메모리 저장 순서대로 처리하는 방법으로, 공간 지역성을 최대화해 캐시 히트율을 높입니다. 반대로 RAM random access는 접근 패턴이 불규칙해 미스가 많아 지연이 큽니다. 예를 들어 Python List보다 Numpy 배열에서 순차 접근이 훨씬 빠릅니다. 이 이미지에서 순차와 랜덤 접근의 성능 차이를 보실 수 있습니다. 랜덤이 왜 문제인지 알 수 있습니다.

PyConKR 2025

Weave with Python

Cache Prefetching

2. Numba와 NumPy로 구현

Pure Python과 Cache Prefetching의 한계

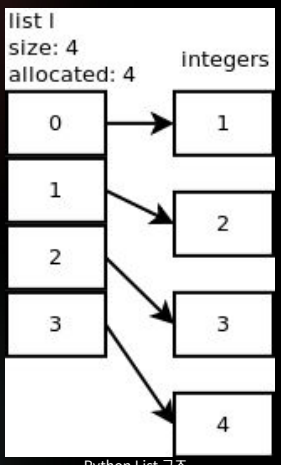
Pure Python: Cache Prefetching 최대활용 불가능

예외: bin, array, typed list

이유: 연속성 데이터 타입 없음

대안: typed List, NumPy Array로 개선

효과: Cache Prefetching 활용가능



Python List 구조

Source : https://stackoverflow.com/questions/3917574/how-is-pythons-list-implemented

python은 동적타입과 인터프리터의 제약으로 인해 연속적인 메모리 관리할 수 있는 데이터 구조를 보통 제공하지 않습니다. 예외로 Binary, bytes, array같은 제한적인 방법이 있습니다. 가장 많이 사용되는 list의 python 구현은 array of point입니다. int같은 primitive데이터 타입도 동일합니다. 이는 point array는 연속성이 있게 되지만, 실제 사용해야하는 데이터는 pointer로 sparse된 메모리 구조를 가지게 됩니다. 이는 cache prefetching을 최대한 활용 못하는 문제로 연결됩니다. 이를 극복하기 위해서 array 데이터구조나 NumPy의 array를 활용 할 수 있습니다.

Cache Prefetching

2. Numba와 NumPy로 구현

예시 테스트

동일 배열 순차 검색, 랜덤 검색 비교 시뮬레이션 테스트

캐싱 방지 목적 데이터크기 조정

유사 케이스: Linked List, Graph

```python
# 데이터 크기 설정
data_size = 100_000_000

# NumPy 배열 생성
data_array = np.arange(data_size, dtype=np.int64)
np.random.seed(77)
np.random.shuffle(data_array)

# 랜덤 인덱스 생성
random_indices = np.arange(data_size, dtype=np.int64)
np.random.seed(42)
np.random.shuffle(random_indices)

# 순차 검색 함수
@njit
def sequential_search(data, target):
    for idx in range(len(data)):
        if data[idx] == target:
            return idx
    return -1

# 랜덤 검색 함수
@njit
def random_search(data, indices, target):
    for idx in indices:
        if data[idx] == target:
            return idx
    return -1
```

Cache prefetching을 검증하기 위해서 테스트를 해보겠습니다. 이 코드는 동일 배열의 순차적 검색과 랜덤 검색을 비교하고 있습니다. 여기서는 꼭 numba가 필요하지 않지만 numpy array를 잘 활용하기 위해서 numba를 사용했습니다. Numpy array대신 python array같은 연속성있는 배열을 사용 할 수 있습니다. Numpy array는 순수 파이썬에서 loop돌릴 시 c코드와 통신의 overhead로 정상적인 테스트를 할 수 없습니다. 데이터 크기도 CPU 캐시보다 충분히 클수 있도록 조정하였습니다. 이 케이스는 실제 활용 중 linked list나 graph의 random access와 유사하다고 할 수 있습니다.

Cache Prefetching

2. Numba와 NumPy로 구현

성능 비교

20배 이상 성능차이

```
ubuntu@ip-172-31-0-222:~/numba-test$ uv run prefetch_test.py
실행 횟수: 5회
순차 검색 평균 시간: 0.035291 ± 0.012906초
랜덤 검색 평균 시간: 0.820182 ± 0.401327초

랜덤 검색은 순차 검색보다 23.24배 느립니다.
```

총 5회 돌려 평균치 한 결과 제 머신에서는 23배 정도의 차이가 나왔습니다. 이와 같이 랜덤 접근과 순차 접근의 성의 차이를 알 수 있습니다.

04

결론

결론

Python의 한계와 극복 방법

순수 Python는 Interpreter, Dynamic Typing 한계 존재

Numba(JIT)로 interpreter 개선
NumPy(or typed.List)로 Dynamic Typing 개선

Branchless, Prefetch 등 기법 추가 CPU 최적화 가능

여기까지 Python CPU최적화 하는 방법들을 소개해드렸습니다. Python은 interpreter와 dynamic typing의 한계가 있습니다. 이를 극복하기 위해서 Numba와 NumPy 혹은 array를 사용해서 개선 할 수 있습니다. 일부 케이스에는 C/C++와 비등한 성능까지 보여줍니다. 추가로 branchless programming과 cache prefetching 기법으로 CPU의 리소스를 최대한 활용 할 수 있는 방법에 대해 말씀을 드렸습니다. 여기 계신 모든 분들의 적용 할 수 있어 의미있는 성능최적화가 되시길 바랍니다.

질문 있으시면 편하게 해주세요 :D