Mojo

Matt Hyatt
Maxwell Sevart

file.mojo
file.🔥 U+1F525 Unicode character)

Background

Founded by :

- Chris Lattner
  - Swift
  - Clang compiler
  - LLVM & MLIR compiler
- Tim Davis
  - Machine Learning thought leader at Google

https://www.timdavis.com/

History of Mojo

There's none!

- September 2022
  - Internally released by Modular Inc.
- May 2023
  - Jupyter Notebooks
- September 2023
  - Linux
- October 19th, 2023
  - OSX

What is Mojo

- Mojo is a language designed **specifically for AI workloads**
- Mojo is optimized for fast performance
  - Especially matrix multiplications
- Looks like python, but leverages principles from rust and c++
- Fully supports python packages

Compiling and Running code

"Mojo code can be ahead-of-time (AOT) or just-in-time (JIT) compiled"

- def main() or fn main()
  - *"fn declaration enforces strongly-typed and memory-safe behaviors"*
- var and let for mutable and immutable types

```
fn main():
    var x: Int = 1
    x += 1
     print(x)
```

1. Create a stand-alone executable with the `build` command:

   ```
   mojo build hello.mojo
   ```

   It creates the executable with the same name as the `.mojo` file, but you can change that with the `-o` option.

2. Then run the executable:

   ```
   ./hello
   ```

Variable Ownership

borrowed:

- The argument was borrowed so it's read only

inout:

- Changes in the function happen outside

owned:

- The function owns a copy of the argument
- Can mutate it without affecting the outer value

```
fn add(borrowed x: Int, borrowed y: Int) -> Int:
    return x + y
```

```
fn add_inout(inout x: Int, inout y: Int) -> Int:
    x += 1
    y += 1
    return x + y
```

```
fn set_fire(owned text: String) -> String:
    text += "🔥"
    return text
```

01. USABILITY *(why use mojo?)*

- **PROGRESSIVE TYPES**
- Your code can be assigned type statically or dynamically.

- dynamic typing allows flexibility,
- conforming to static types allows Mojo to compile your code for optimal speed

- **ZERO COST ABSTRACTIONS**
- Usually class abstractions come at the price of slower code, since pointers are used to refer to other data structures *(sometimes excessively)*

- Mojo gives classes and structs allocate memory locally within the class/struct to reduce space and time burdens.

01. USABILITY *(why use mojo?)*

- **OWNERSHIP + BORROW CHECKER**
- Similar to Rust, Mojo maintains a concept of data ownership.
- Checking for ownership at compile-time reduces memory errors ie: null pointer dereferencing

```
def reorder_and_process(owned x: HugeArray):
  sort(x)        # Update in place

  give_away(x^) # Transfer ownership

  print(x[0])   # Error: 'x' moved away!
```

```
def exp[dt: DType, elts: Int]
    (x: SIMD[dt, elts]) -> SIMD[dt, elts]:
  x = clamp(x, -88.3762626647, 88.37626266)
  k = floor(x * INV_LN2 + 0.5)
  r = k * NEG_LN2 + x
  return ldexp(_exp_taylor(r), k)
```

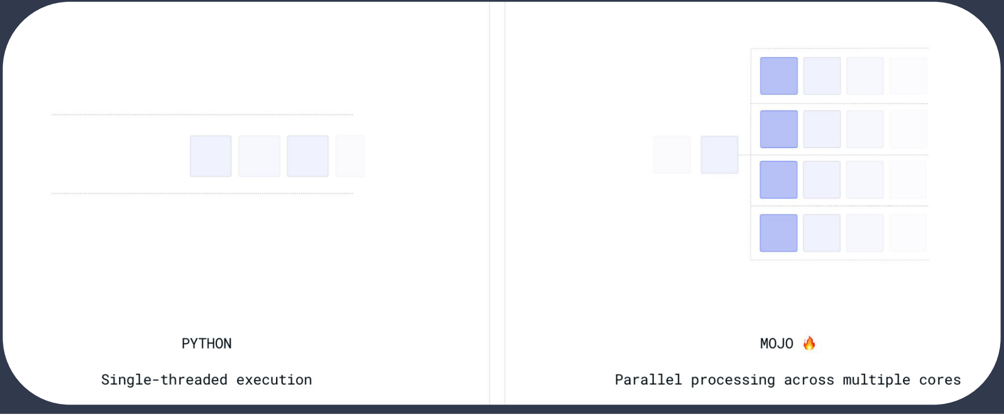
- **PORTABLE PARAMETRIC ALGORITHMS**
- Mojo lets you define parametric functions that can operate on different types
- Moreover, the compiler will optimize them
  - *meta-programming*
- Mojo SIMD data structures allow instructions to be executed in parallel across all data in the structure
  - This is huge! Especially while staying agnostic to hardware

01. USABILITY *(why use mojo?)*

- **LANGUAGE INTEGRATED AUTO-TUNING**
- Given a range of values, Mojo will find the one that yields the best performance

```
def exp_buffer[dt: DType](data: ArraySlice[dt]):

  # Search for the best vector length
  alias vector_len = autotune(1, 4, 8, 16, 32)

  # Use it as the vectorization length
  vectorize[exp[dt, vector_len]](data)
```

02. Mojo is **FAST**



LANGUAGES	TIME (S) *	SPEEDUP VS PYTHON
PYTHON 3.10.9	1027 s	1x
PYPY	46.1 s	22x
SCALAR C++	0.20 s	5000x
MOJO 🔥	0.03 s	35000x

68,000x speedup on certain machines

02. PERFORMANCE *(why use mojo?)*

Parallel heterogeneous runtime

- This feature allows the developer to spread work across many machines
- Even if the machines are different!

Multi-Level Intermediate Representation (MLIR)

- Used to build tensorflow compilers
- Standardize infrastructure and building blocks
  - Even across different hardware

Conforming to these standards is what allows mojo to be executed in parallel.

Basic Code Examples

Basic Arithmetic Operation:

```
fn add(x: Int, y: Int) -> Int:
     return x + y
```

Variable Declaration:

```
fn main():
    let x = 1

    let y: Int
    y = 1

    var z = 0
    z += 1
```

03. INTEROPERABILITY

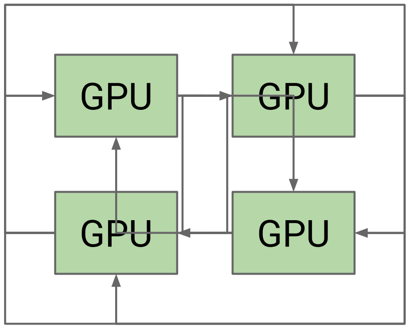
Because Mojo conforms to Python syntax and standards (even if only at face value), users can leverage the entire python ecosystem

```
from PythonInterface import Python

np = Python.import_module("numpy")

array = np.array([1, 2, 3])
print(array)
```

04. EXTENSIBILITY

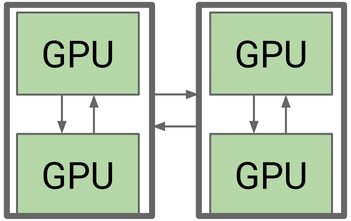
Graph Rewrites

- Rewriting and controlling computation can make it more efficient



4*3 = 12 communications

6 communications



Kernel Fusion

- Native Mojo allows developers to fuse several computation operations into a single operation. This reduces redundant memory access and the unnecessary creation of kernels *(a job sent to the hardware accelerator).*

H(G(F(A))) + H(G(F(B))) → HGF(A) + HGF(B)

Limitations Surprises

The language is still being designed. Several important features are currently not implemented **yet**:

- No recursion
- No lifetime tracking inside collections
- No polymorphism
- No async for or async with
- No parametric aliases
- No lambda syntax
- No list or dict comprehensions

“we need to get the core language semantics nailed down before adding ___”

Cool Features

- Native types
  - tensor
- Native modules
  - Polynomial
  - Limit (infinite numbers)
  - Info (OS and machine hardware/instructions)
  - Intrinsics (instruction prefetch / data locality)
  - Coroutine (paused execution)
  - Benchmark
  - reduction.map_reduce

On this page

`is_x86`
`has_sse4`
`has_avx`
`has_avx2`
`has_avx512f`
`has_avx512_vnni`
`has_neon`
`is_apple_m1`

`PrefetchLocality`

The prefetch locality.

The locality, rw, and cache type correspond to LLVM prefetch intrins

Aliases:

- `NONE = __init__(0)`: No locality.
- `LOW = __init__(1)`: Low locality.
- `MEDIUM = __init__(2)`: Medium locality.
- `HIGH = __init__(3)`: Extremely local locality (keep in cache).

Conclusion

Q&A

- What is the hardest part about
- What is <> used for

NOTES

- Keynote @ 34:00 is very interesting
- Roadmap shows some of the challenges the team is facing when **actively** building a new language

Create a Modular account to get started

Once your email is verified you'll be able to download and install Modular products. We suggest signing up with your work email.

- Not open-source yet… not fully supported yet