Numba-MLIR

MLIR-based python compiler

Ivan Butygin, Diptorup Deb, 2023

Overview

- MLIR-based JIT compiler for numerical python code
- Same programming API as Numba
- Based on Numba frontend and type inference
- Codegen for CPUs and GPUs through MLIR
  - Numpy parallelization and offload
  - cuda.jit-like kernel API

Why Numba-MLIR

- MLIR already have diverse set of 'dialects'
- Portability
  - reuse code between CPU and GPU, as well between different GPU vendors
- Reuse code across multiple MLIR projects and domains
  - Linalg dialect and optimizations
  - GPU dialect and lowering
  - Loop optimizations
    - Fusion
    - Tiling
- Extending to other programming models
  - JIT-compiled numpy-like library
  - Distributed numpy

Numba-MLIR: Current status

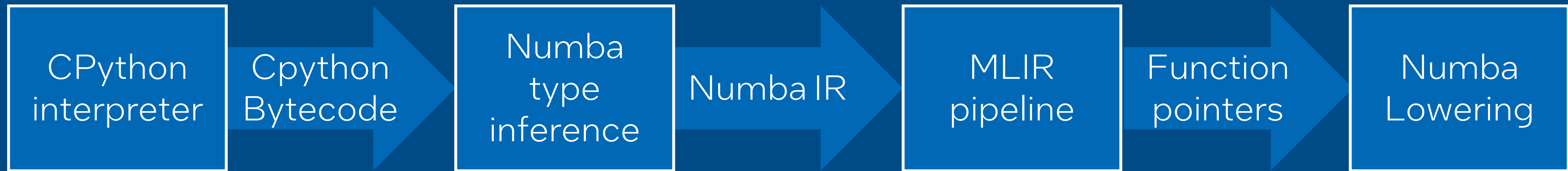
- Python math, complex, loops, arbitrary control flow
- Numpy arrays and funcs
  - Ufuncs
  - Various array indexing modes
  - Broadcasting
  - Implementing new functions is straightforward (no C++ code involved)
- Numba.prange
- GPU code generation for Intel devices
- CUDA should be relatively straightforward to add

Numba-MLIR programming API

```
import numpy as np
import numba_mlir

a = np.ones(42, dtype=np.float32)
b = np.ones(42, dtype=np.float32)

@numba_mlir.njit(parallel=True)
def foo(a, b):
    return np.sum(a + b)

res = foo(a, b)
```

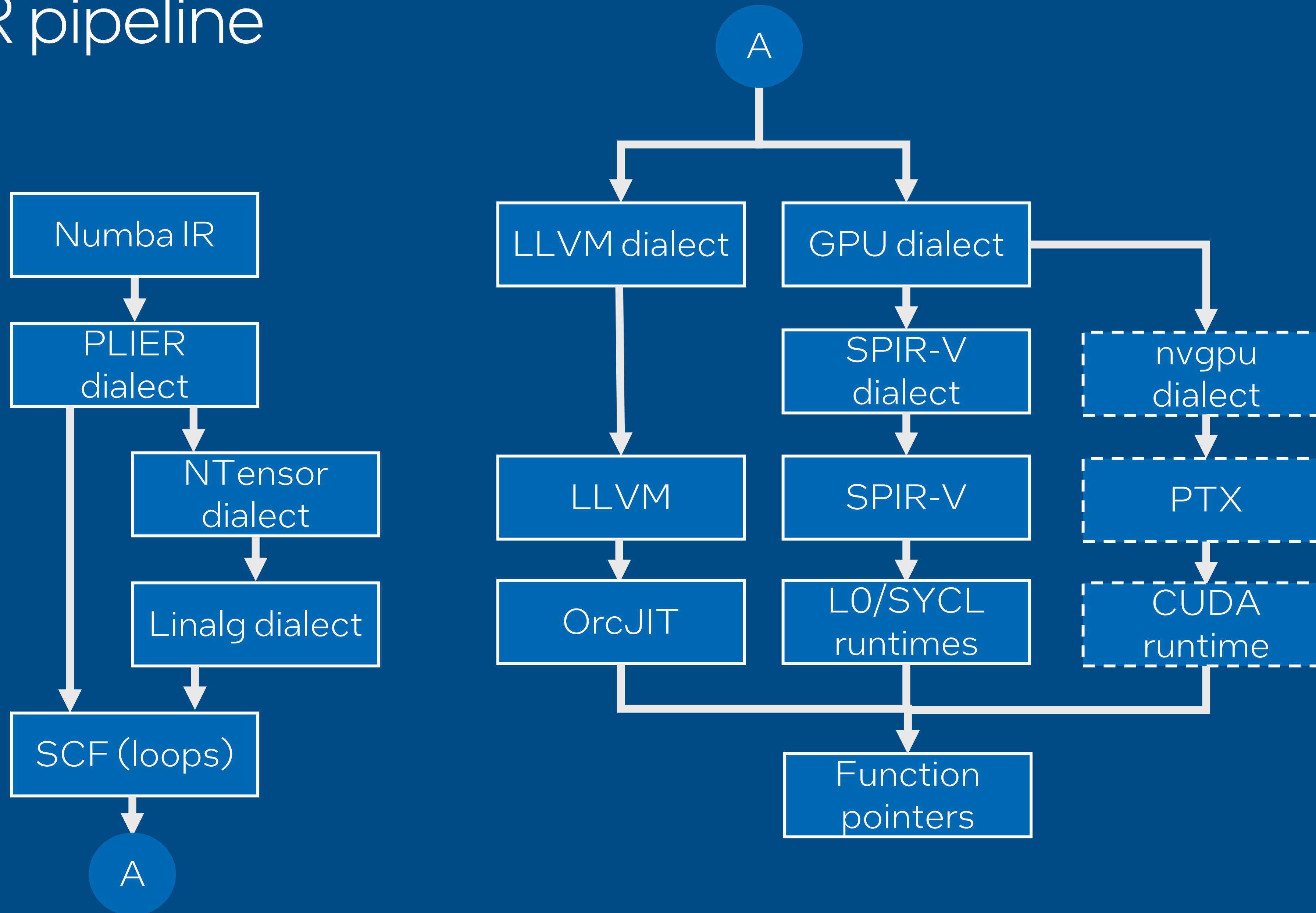
```
import numpy as np
import numba
import numba_mlir

a = np.ones(42, dtype=np.float32)
b = np.ones(42, dtype=np.float32)

@numba_mlir.njit(parallel=True)
def foo(a, b):
    c = 0.0
    for i in numba.prange(a.shape[0]):
        c += a[i] + b[i]
    return c

res = foo(a, b)
```

Numba-MLIR Pipeline



- numba_mlir.njit == numba.njit with custom pipeline
- Replaces all passes after type inference with custom MLIR pass
- Custom pass run entire MLIR pipeline inside
- Final code converted to llvm and compiled into native using OrcJIT, func pointer passed back to numba
  - To avoid interop issues (numba is llvm14, we are llvm HEAD)

MLIR pipeline



Case study: supporting broadcast for numpy parallel operations

```
@njit(parallel=True)
def foo(a, b):
    return a + b

a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3]])

result = foo(a, b)
```

Case study: Broadcasting

- MLIR optimization pass for broadcasting
  - Reuse a lot of existing MLIR infra
  - Array static shapes propagation
  - Integer range analysis
  - Constant folding
  - Control flow simplifications
- Works on CPU and GPU
- Isolated from numba or numpy implementation details

Overload API

- Intended to emit high-level linalg primitives if possible
- Original Numba overloads are still needed
  - Numba-MLIR still uses type inference from Numba
  - Actual overload body is not used, only types
- Custom overload API for actual function impl
- Can call arbitrary native functions
- Can inline arbitrary code, including `prange` and numpy calls

Overload example

```
@register_func("numpy.dot", numpy.dot, out="out")
def _linalg_matmul2d(builder, a, b):
    shape1 = a.shape
    shape2 = b.shape
    ...
    iterators = ["parallel", "parallel", "reduction"]
    expr1 = "(d0,d1,d2) -> (d0,d2)"
    expr2 = "(d0,d1,d2) -> (d2,d1)"
    expr3 = "(d0,d1,d2) -> (d0,d1)"
    maps = [expr1, expr2, expr3]
    res_shape = (shape1[0], shape2[1])
    dtype = broadcast_type_arrays(builder, (a, b))
    init = builder.init_tensor(res_shape, dtype, 0)

    def body(a, b, c):
        return a * b + c

    return builder.linalg_generic((a, b), init, iterators, maps, body)
```

Alternative parfor lowering

- Instead of lowering entire function lower only specific parfor nodes
- No frontend part, no custom overloads
- Utilize MLIR-based GPU lowering and runtime
- Can be plugged into existing pipeline with minimal changes
- High-level opts (loops fusion) handled by existing Numba/ParallelAccelerator

How to integrate with Numba?

- Compatible with ongoing Numba frontend redesign
- Incremental integration with Numba
  - Separating different optimizations into dedicated Numba passes
  - NumbaIR->MLIR->NumbaIR roundtrip
  - Alternative Parfor lowering pipeline
- Redesigned overload API
  - Numba-MLIR overloads are generally more high-level than Numba ones
    - Intended to emit linalg ops but we can emit loops directly
  - Numba-MLIR -> Numba overloads shim is doable

Try it out

- https://github.com/numba/numba-mlir
- **`conda install numba-mlir -c dppy/label/dev -c intel -c conda-forge -c numba`**

Backup

Case study 1: Broadcasting

- Initial naïve implementation
  - Expands each axis for each arg individually
  - Multiple data copies, no fusion
- Separate JIT func specializations for unit and non-unit array dims
- MLIR optimization pass for broadcasting
  - Reuse a lot of existing MLIR infra
  - Array static shapes propagation
  - Integer range analysis
  - Constant folding
  - Control flow simplifications
  - Isolated from numba or numpy implementation details

Broadcasting

```
%dim0 = dim %arg, 0
%dim1 = dim %arg, 1
%0 = scf.if (%dim0 == 1) {
    %1 = expand axis 0 %arg
    yield %1
} else  {
    yield %arg
}

%2 = scf.if (%dim1 == 1) {
    %3 = expand axis 1 %0
    yield %3
} else  {
    yield %0
}
use %2
```

GPU offload

GPU offloading

GPU offloading is supported via dpctl library

- Pass dpctl arrays as inputs to JIT function instead of numpy arrays
- Numpy functions operating on dpctl arrays inside jitted function will be offloaded to gpu
- Compute follows data: if numpy function inputs are on specific device, output will be also on this device
- Only Intel devices are supported for now

GPU offloading

```
import numpy as np
import numba_mlir
import dpctl.tensor as dpt

a = dpt.ones(42, dtype=np.float32, device="level_zero:gpu:0")
b = dpt.ones(42, dtype=np.float32, device="level_zero:gpu:0")

@numba_mlir.njit (parallel=True)
def foo(a, b):
    return np.sum(a + b) # Will be offloaded to gpu

res = foo(a, b)
```

GPU offloading

```
import numpy as np
import numba_mlir
import dpctl.tensor as dpt

a = dpt.ones(42, dtype=np.float32, device="level_zero:gpu:0")
b = dpt.ones(42, dtype=np.float32, device="level_zero:gpu:0")

@numba_mlir.njit(parallel=True)
def foo(a, b):
    c = 0.0
    for i in numba.prange(a.shape[0]): # Will be offloaded to gpu
        c += a[i] + b[i]
    return c

res = foo(a, b)
```

GPU offload: kernel style programming

```
import dpctl.tensor as dpt
from numba_mlir.kernel import kernel, get_global_id, DEFAULT_LOCAL_SIZE

@kernel
def foo(a, b, c):
    i = get_global_id(0)
    j = get_global_id(1)
    k = get_global_id(2)
    c[i] = a[i] + b[i]

a = dpt.ones((5,6,7))
b = dpt.ones((5,6,7))
c = dpt.empty((5,6,7))
global_size = a.shape
local_size = (2,3,4) # Or DEFAULT_LOCAL_SIZE to let runtime choose

foo[global_size, local_size](a, b, c)
```

GPU offload: kernel style programming

- OpenCL-style programming model
- Features:
  - Numpy-style multidim arrays and slices
  - Barriers and fences
  - Local and private arrays
  - Atomic ops
  - Group reduction ops
  - Global size not needed to be divisible by local size
  - Automatic fp64 truncation for hw which doesn't support it
  - Software simulator

FP64 truncation

- Truncates all fp64 ops to fp32
- Possible options are **`True/False/"auto"`** (default if **`False`**)
- Works both for kernel and njit offload
  - **`@numba_mlir.njit(gpu_fp64_truncate="auto")`**
  - **`@numba_mlir.kernel.kernel(gpu_fp64_truncate="auto")`**

GPU kernel simulator

- Runs entire kernel in interpreted mode
- No compilation or type inference is done
- Single-threaded and slow
- Barriers and group ops are supported (via coroutines from greenlet package)

```
from numba_mlir.kernel import kernel_sim

@kernel_sim
def foo(a, b, c):
    i = get_global_id(0)
    j = get_global_id(1)
    k = get_global_id(2)
    c[i] = a[i] + b[i]
```

CUDA support

- Device code
  - Current pipeline is GPU dialect->SPIR-V
  - Investigate GPU->NVGPU pipeline from upstream
- Host code
  - Current runtime only supports L0 and OpenCL backends
  - SYCL itself supports CUDA backend, our runtime impl can be extended
  - --OR-- Write standalone runtime
- Type inference
  - Need some array type as entry point
  - Only basic type inference hooks are needed, can reuse existing numpy overloads
- Other
  - All linalg and loops opts will work out of the box
  - F64 emulation will work out of the box
  - Kernel sim will work out of the box

Extending

Extending: linalg_generic

- Main linalg primitive
- Mapped directly to `MLIR` **`linalg.generic`** op
- Format: **`builder.linalg_generic(inputs, outputs, iterators, maps, body)`**
- Arrays are immutable
- Inputs – input arrays, will only be read from
- Outputs – output arrays, used to determine output shape and initial values for reductions
- Iterator – enclosing loops types, can be either "parallel" or "reduction"
- Maps – mappings from iterators space to array indices
- Body – body for single iteration, accepts single elements from each input and output arrays, according to iterators and mapping, returns new element vales for corresponding out array
- Returns new immutable arrays same shape and type as outputs

Extending: linalg_generic indices access

```
@register_func("numpy.eye", numpy.eye)
def eye_impl(builder, N, M=None, k=0, dtype=None):
    …
    init = builder.init_tensor((N, M), dtype)
    idx = builder.from_elements(k, builder.int64)

    iterators = ["parallel"] * 2
    maps = ["(d0, d1) -> (0)", "(d0, d1) -> (d0, d1)"]

    def body(a, b):
        i = _linalg_index(0)
        j = _linalg_index(1)
        return 1 if (j - i) == a else 0

    return builder.linalg_generic(idx, init, iterators, maps, body)
```

Extending: external calls

```
def _mkl_gemm(builder, a, b, alpha, beta, shape1, shape2):
    copy_a = _check_mkl_strides(a)
    copy_b = _check_mkl_strides(b)

    a = builder.ifop(copy_a, lambda: builder.force_copy(a), lambda: a)
    b = builder.ifop(copy_b, lambda: builder.force_copy(b), lambda: b)

    dtype = a.dtype
    func_name = f"mkl_gemm_{dtype_str(builder, dtype)}"
    device_func_name = func_name + "_device"

    res_shape = (shape1[0], shape2[1])
    c = builder.init_tensor(res_shape, dtype)

    return builder.external_call(
        func_name, (a, b), c, attrs={"device_func": device_func_name}
    )
```

Extending: inlining code

```
def func(a, ind):
    s = a.size
    res = numpy.empty((s,), a.dtype)
    curr = 0
    for i in range(s):
        if ind[i]:
            res[curr] = a[i]
            curr += 1
    return res[0:curr]

return builder.inline_func(func, arr.type, arr, index)
```

intel®