Numba-MLIR

MLIR-based python compiler

Ivan Butygin, Diptorup Deb, 2023

intel®

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
  - Numpy parallelization and offload
  - cuda.jit-like kernel API

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 (parallel loops)
- 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)
```

Kernel-style programming API

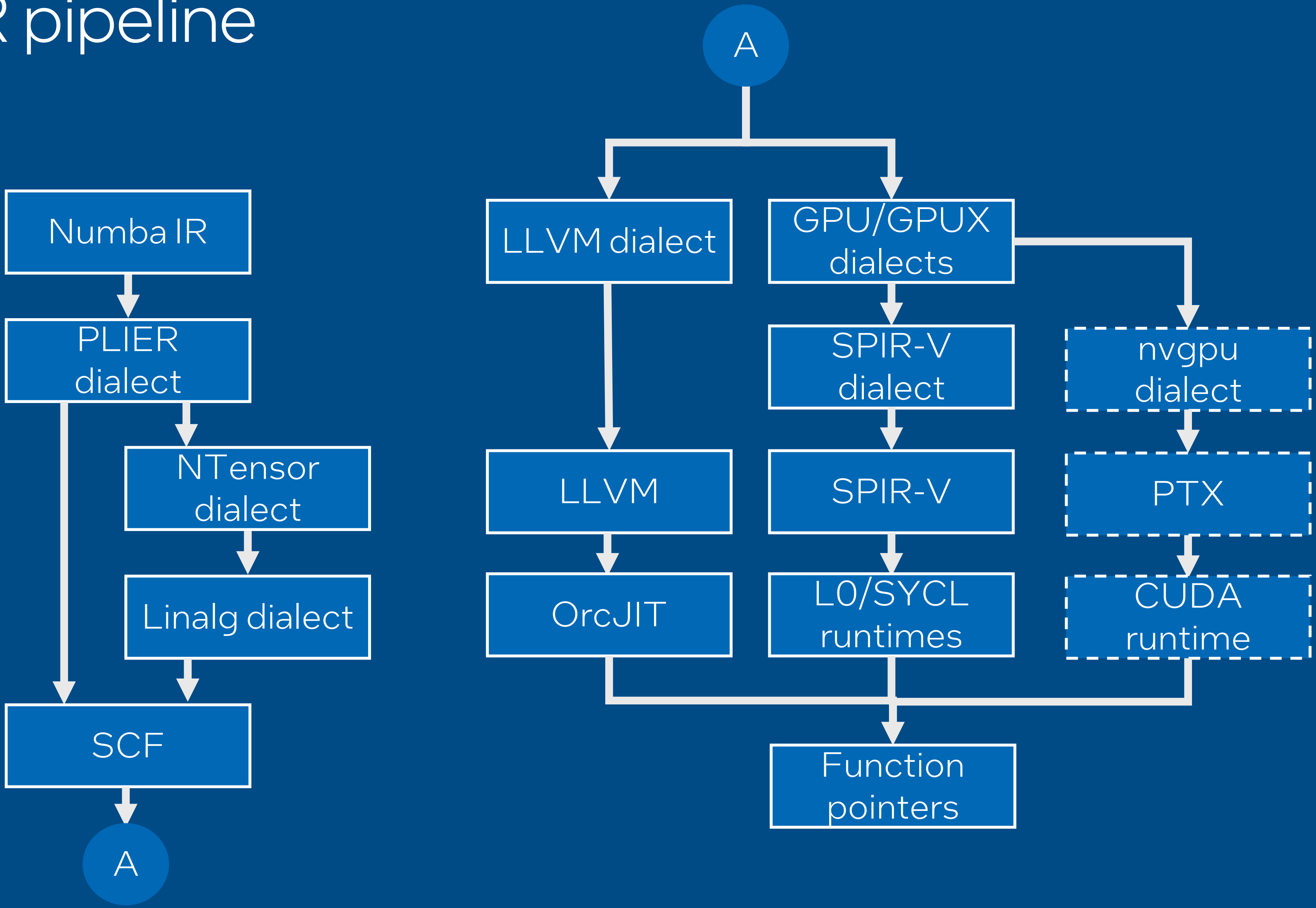
```
import dpctl.tensor as dpt
from numba_mlir.kernel import kernel, get_global_id, DEFAULT_LOCAL_SIZE

@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)
```

MLIR pipeline



Dialects

- PLIER (Numba IR dialect)
  - Frontend dialect, 1:1 conversion from Numba IR
  - Generally, not intended for optimizations
- NTensor
  - Numpy dialect
  - Mutable tensors
  - Array API compute-follows-data/device offload
- GPUX/GPU_RUNTIME
  - GPU dialect extensions

NTensor: New tensor dialect motivation

- Mutable tensors
  - Memref is too low level
  - Array setitem
  - Numpy out args
- ArrayAPI compute-follows-data model support
- Broadcasts on dynamic shapes

NTensor type

- Type for Numpy arrays
- Derived from **`ShapedType`** with usual interface
- Always ranked
- Format: **`!ntensor.ntensor<shapeXtype[, layout[, device]]>`**
  - Shape and type – both static and dynamic shapes are supported
  - Layout – “C” or “A”, deprecated and will be removed
  - Device – used for offload
- **`!ntensor.ntensor<?x5xf32 ,”C”, #device<gpu>>`**

NTensor ops

- Ops are separated into 2 categories:
- High level ops
  - Intended to be emitted by the frontends
  - Numpy call ops support named and omitted parameters
  - Support out parameters
  - Array ops support fancy indexing
- Low level ops
  - All params are positional
  - Array indexing is explicit and load/store/subview ops follow tensor/memref semantics (e.g., out of bounds access is UB)
- Doesn’t hardcode specific set of ops

Array indexing

```
@njit(parallel=True)
def foo(a):
    a[1:2] = 1

module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo(%arg0: !ntensor.ntensor<?xsi32 : "C">) -> none {
    %c1_i64 = arith.constant 1 : i64
    %c2 = arith.constant 2 : index
    %c1 = arith.constant 1 : index
    %0 = numba_util.undef : none
    %1 = numba_util.sign_cast %c1_i64 : i64 to si64
    %2 = "plier.build_slice"(%c1, %c2, %0) : (index, index, none) -> !plier.slice
    "plier.setitem"(%arg0, %2, %1) : (!ntensor.ntensor<?xsi32 : "C">, !plier.slice, si64) ->
()
    return %0 : none
  }
}
```

Array indexing

```
module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo(%arg0: !ntensor.ntensor<?xsi32 : "C">) -> none {
    %c1_i64 = arith.constant 1 : i64
    %c2 = arith.constant 2 : index
    %c1 = arith.constant 1 : index
    %0 = numba_util.undef : none
    %1 = numba_util.sign_cast %c1_i64 : i64 to si64
    %2 = ntensor.build_slice(%c1 : %c2 : )
    %c1_i32 = arith.constant 1 : i32
    %3 = numba_util.sign_cast %c1_i32 : i32 to si32
    ntensor.setitem(%arg0 : !ntensor.ntensor<?xsi32 : "C">) [%2 :
!ntensor.slice] = (%3 : si32)
    return %0 : none
  }
}
```

Array indexing

```
module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo(%arg0: !ntensor.ntensor<?xsi32 : "C">) -> none {
    %c0 = arith.constant 0 : index
    %c1_i32 = arith.constant 1 : i32
    %c2 = arith.constant 2 : index
    %c1 = arith.constant 1 : index
    %0 = numba_util.undef : none
    %1 = ntensor.build_slice(%c1 : %c2 : )
    %2 = numba_util.sign_cast %c1_i32 : i32 to si32
    %dim = ntensor.dim %arg0, %c0 : !ntensor.ntensor<?xsi32 : "C">
    %begin, %end, %step, %count = ntensor.resolve_slice %1, %dim
    %3 = ntensor.subview %arg0[%begin] [%count] [%step] : !ntensor.ntensor<?xsi32 : "C"> to
!ntensor.ntensor<?xsi32 : "C">
    %dim_0 = ntensor.dim %3, %c0 : !ntensor.ntensor<?xsi32 : "C">
    %4 = ntensor.create(%dim_0) = (%2 : si32) : !ntensor.ntensor<?xsi32 : "C">
    ntensor.copy %4, %3 : !ntensor.ntensor<?xsi32 : "C"> to !ntensor.ntensor<?xsi32 : "C">
    return %0 : none
  }
}
```

Array indexing

```
module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo(%arg0: memref<?xi32> ) -> none {
    ...
    %0 = numba_util.undef : none
    %dim = memref.dim %arg0, %c0 : memref<?xi32>
    %1 = arith.cmpi sle, %dim, %c2 : index
    %2 = arith.select %1, %dim, %c2 : index
    %3 = arith.subi %2, %c1 : index
    %subview = memref.subview %arg0[1] [%3] [1] : memref<?xi32> to memref<?xi32, strided<[1], offset: 1>>
    %4 = tensor.empty(%3) : tensor<?xi32>
    %5 = linalg.fill ins(%c1_i32 : i32) outs(%4 : tensor<?xi32>) -> tensor<?xi32>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]} ins(%5 : tensor<?xi32>) outs(%subview : memref<?xi32, strided<[1], offset: 1>>) {
    ^bb0(%in: i32, %out: i32):
      linalg.yield %in : i32
    }
    return %0 : none
  }
}
```

NTensor Ops

```
@njit
def foo():
    return np.arange(10)
```

```
module {
  func.func @foo() -> !ntensor.ntensor<?xsi64 : "C"> {
    %c10_i64 = arith.constant 10 : i64
    %0 = numba_util.sign_cast %c10_i64 : i64 to si64
    %1 = ntensor.call "numpy.arange"  (%0) : si64 -> !ntensor.ntensor<?xsi64 : "C">
    return %1 : !ntensor.ntensor<?xsi64 : "C">
  }
}
```

NTensor Ops

```
module {
  func.func @foo() -> !ntensor.ntensor<?xsi64 : "C"> {
    %c10_i64 = arith.constant 10 : i64
    %0 = numba_util.sign_cast %c10_i64 : i64 to si64
    %1 = numba_util.undef : none
    %2 = numba_util.undef : none
    %3 = numba_util.undef : none
    %4 = ntensor.primitive "numpy.arange" (%0, %1, %2, %3) : si64, none, none,
none -> !ntensor.ntensor<?xsi64 : "C">
    return %4 : !ntensor.ntensor<?xsi64 : "C">
  }
}
```

NTensor Ops

```
module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo() -> memref<?xi64> {
    %cst = arith.constant dense<1> : tensor<1xi64>
    %cst_0 = arith.constant dense<0> : tensor<1xi64>
    %0 = tensor.empty() : tensor<10xi64>
    %1 = linalg.generic {indexing_maps = [#map, #map, #map1], iterator_types = ["parallel"]} ins(%cst_0, %cst
: tensor<1xi64>, tensor<1xi64>) outs(%0 : tensor<10xi64>) {
    ^bb0(%in: i64, %in_1: i64, %out: i64):
      %3 = linalg.index 0 : index
      %4 = arith.index_cast %3 : index to i64
      %5 = arith.muli %in_1, %4 : i64
      %6 = arith.addi %in, %5 : i64
      linalg.yield %6 : i64
    } -> tensor<10xi64>
    %2 = bufferization.to_memref %1 : memref<10xi64>
    %cast = memref.cast %2 : memref<10xi64> to memref<?xi64>
    return %cast : memref<?xi64>
  }
}
```

Out args

```
@njit
def foo(a, b, c):
    np.add(a, b, out=c)
```

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?xsi32 : "C">, %arg1: !ntensor.ntensor<?xsi32 : "C">, 
%arg2: !ntensor.ntensor<?xsi32 : "C">) -> none {
    %0 = numba_util.undef : none
    %1 = ntensor.call "numpy.add"  (%arg0, %arg1, out:%arg2) : !ntensor.ntensor<?xsi32 : "C">, 
!ntensor.ntensor<?xsi32 : "C">, !ntensor.ntensor<?xsi32 : "C"> -> !ntensor.ntensor<?xsi32 : 
"C">
    return %0 : none
  }
}
```

Out args

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?xsi32 : "C">, %arg1: !ntensor.ntensor<?xsi32 : "C">,
%arg2: !ntensor.ntensor<?xsi32 : "C">) -> none {
    %0 = numba_util.undef : none
    %1 = ntensor.primitive "numpy.add" (%arg0, %arg1) : !ntensor.ntensor<?xsi32 : "C">,
!ntensor.ntensor<?xsi32 : "C"> -> !ntensor.ntensor<?xsi32 : "C">
    ntensor.copy %1, %arg2 : !ntensor.ntensor<?xsi32 : "C"> to !ntensor.ntensor<?xsi32 : "C">
    return %0 : none
  }
}
```

Out args

```
module attributes {numba.pipeline_jump_markers = []} {
  func.func @foo(%arg0: memref<?xi32>, %arg1: memref<?xi32>, %arg2: memref<?xi32>) -> none {
    %0 = numba_util.undef : none
    %1 = bufferization.to_tensor %arg0 : memref<?xi32>
    %2 = bufferization.to_tensor %arg1 : memref<?xi32>
    linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%1, 
%2 : tensor<?xi32>, tensor<?xi32>) outs(%arg2 : memref<?xi32>) {
    ^bb0(%in: i32, %in_0: i32, %out: i32):
      %3 = arith.extsi %in : i32 to i64
      %4 = arith.extsi %in_0 : i32 to i64
      %5 = arith.addi %3, %4 : i64
      %6 = arith.trunci %5 : i64 to i32
      linalg.yield %6 : i32
    }
    return %0 : none
  }
}
```

Lowering

- Lower to linalg-on-tensor if can prove it safe
  - No writes to output
  - LocalAliasAnalysis
- Copy ops lowered to mixed linalg, with memref output
- Use elementwise ops fusion patterns from upstream
- Lower everything else to memref
- Lowering defined in python

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)
```

Bufferization

- Mixed tensor/memref – cannot use oneshot
- Mostly legacy bufferization with some custom patterns
- **`ChangeLayoutOp`** – to avoid unnecessary copies

Bufferization

- Use legacy, dialect-conversion patterns
- When layout aren't compatible insert **`ChangeLayoutOp`** instead of copy
- **`ChangeLayoutOp`** have set of patterns to propagate itself through ops
- Remaining will **`ChangeLayoutOp`** be converted to copy ops
- So, we can still optimistically assume identity layout but downgrade to strided when needed
- All transformations are local, don't need complex analysis

Bufferization

```
%1 = memref.subview ... : memref<?xi32, strided>
%2 = numba_util.change_layout %1 memref<?xi32, strided> to
memref<?xi32>
%3 = memref.load %2 : memref<?xi32>
```

to

```
%1 = memref.subview ... : memref<?xi32, strided>
%2 = memref.load %1 : memref<?xi32, strided>
```

Broadcasting

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

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?x?xsi32 : "C">, %arg1:
!ntensor.ntensor<?x?xsi32 : "C">) -> !ntensor.ntensor<?x?xsi32 : "C">
{
    %0 = ntensor.primitive "operator.add" (%arg0, %arg1) :
!ntensor.ntensor<?x?xsi32 : "C">, !ntensor.ntensor<?x?xsi32 : "C"> ->
!ntensor.ntensor<?x?xsi32 : "C">
    return %0 : !ntensor.ntensor<?x?xsi32 : "C">
  }
}
```

Broadcasting

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?x?xsi32 : "C">, %arg1: !ntensor.ntensor<?x?xsi32 :
"C">) -> !ntensor.ntensor<?x?xsi32 : "C"> {
    %0:2 = ntensor.broadcast(%arg0, %arg1) : !ntensor.ntensor<?x?xsi32 : "C">,
!ntensor.ntensor<?x?xsi32 : "C"> -> !ntensor.ntensor<?x?xsi32 : "C">,
!ntensor.ntensor<?x?xsi32 : "C">
    %1 = ntensor.to_tensor %0#0 : !ntensor.ntensor<?x?xsi32 : "C"> to tensor<?x?xsi32>
    %2 = ntensor.to_tensor %0#1 : !ntensor.ntensor<?x?xsi32 : "C"> to tensor<?x?xsi32>
    %3 = ntensor.create(%dim, %dim_0) : !ntensor.ntensor<?x?xsi32>
    %4 = ntensor.to_tensor %3 : !ntensor.ntensor<?x?xsi32> to tensor<?x?xsi32>
    %5 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel",
"parallel"]} ins(%1, %2 : tensor<?x?xsi32>, tensor<?x?xsi32>) outs(%4 : tensor<?x?xsi32>) {
    ^bb0(%in: si32, %in_1: si32, %out: si32):
      ...
      linalg.yield %10 : si32
    } -> tensor<?x?xsi32>
    %6 = ntensor.from_tensor %5 : tensor<?x?xsi32> to !ntensor.ntensor<?x?xsi32 : "C">
    return %6 : !ntensor.ntensor<?x?xsi32 : "C">
  }
}
```

Broadcasting: naïve approach

- Broadcast every dim of every arg individually
- Functionally correct
- Potentially multiple copies per each arg
- No fusion possible

Broadcasting: naïve approach

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

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

Broadcasting: naïve approach

```
#map = affine_map<(d0, d1) -> (0, d1)>
#map1 = affine_map<(d0, d1) -> (d0, d1)>

%0 = ntensor.to_tensor %arg0 : !ntensor.ntensor<?x?xsi32 : "C"> to tensor<?x?xsi32>
...
%13 = scf.if %12 -> (tensor<?x?xsi32>) {
  %31 = tensor.empty(%9, %dim_4) : tensor<?x?xsi32>
  %32 = linalg.generic {indexing_maps = [#map, #map1], iterator_types = ["parallel",
"parallel"]} ins(%0 : tensor<?x?xsi32>) outs(%31 : tensor<?x?xsi32>) {
  ^bb0(%in: si32, %out: si32):
    linalg.yield %in : si32
  } -> tensor<?x?xsi32>
  scf.yield %32 : tensor<?x?xsi32>
} else {
  scf.yield %0 : tensor<?x?xsi32>
}
```

Broadcasting: improvements

- Specialize into separate functions for unit-size and non-unit-size input array args
  - For unit-size shapes broadcasting is trivially folded
- ShapeRangePropagation
  - Extends **`IntegerRangeAnalysis`** for shaped types (memref, tensor, ntensor)
  - Attach shape range to function args

```
%arg0: !ntensor.ntensor<?x?xsi32 : "C"> {numba.shape_range =
[#numba_util.index_range<[2, 9223372036854775807]>, #numba_util.index_range<[2,
9223372036854775807]>]}
```

GPU pipeline

GPU pipeline

```
import dpctl.tensor as dpt
a = dpt.ones((10,10))

@njit(parallel=True)
def foo(a):
    return a + 1
```

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?x?xsi32 : "C", #gpu_runtime.region_desc<device = 
"level_zero:gpu:0">>) -> !ntensor.ntensor<?x?xsi64 : "C"> {
    %1 = "plier.const"() {val = 1 : si64} : () -> si64
    %2 = "plier.binop"(%0, %1) {op = "+"} : (!ntensor.ntensor<?x?xsi32 : "C", #gpu_runtime.region_desc<device
= "level_zero:gpu:0">>, si64) -> !ntensor.ntensor<?x?xsi64 : "C">
    %3 = "plier.cast"(%2) : (!ntensor.ntensor<?x?xsi64 : "C">) -> !ntensor.ntensor<?x?xsi64 : "C">
    return %3 : !ntensor.ntensor<?x?xsi64 : "C">
  }
}
```

GPU pipeline: env propagation

- **`PropagateEnvironmentPass`**
  - Initially, only func args have env attached
  - Propagates env through ntensor ops
  - Based on DataflowAnalysis framework
  - Will emit error if op args have incompatible env (different devices)

GPU pipeline: env propagation

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?x?xsi32 : "C", #gpu_runtime.region_desc<device =
"level_zero:gpu:0">> {numba.restrict}) -> !ntensor.ntensor<?x?xsi64 : "C"> {
    %c1_i64 = arith.constant 1 : i64
    %0 = numba_util.sign_cast %c1_i64 : i64 to si64
    %1 = ntensor.primitive "operator.add" (%arg0, %0) : !ntensor.ntensor<?x?xsi32 : "C",
#gpu_runtime.region_desc<device = "level_zero:gpu:0">>, si64 -> !ntensor.ntensor<?x?xsi64 : "C">
    return %1 : !ntensor.ntensor<?x?xsi64 : "C">
  }
}
```

```
module {
  func.func @foo(%arg0: !ntensor.ntensor<?x?xsi32 : "C", #gpu_runtime.region_desc<device =
"level_zero:gpu:0">> {numba.restrict}) -> !ntensor.ntensor<?x?xsi64 : "C"> {
    %c1_i64 = arith.constant 1 : i64
    %0 = numba_util.sign_cast %c1_i64 : i64 to si64
    %1 = ntensor.primitive "operator.add" (%arg0, %0) : !ntensor.ntensor<?x?xsi32 : "C",
#gpu_runtime.region_desc<device = "level_zero:gpu:0">>, si64 -> !ntensor.ntensor<?x?xsi64 : "C",
#gpu_runtime.region_desc<device = "level_zero:gpu:0">>
    %2 = ntensor.cast %1 : !ntensor.ntensor<?x?xsi64 : "C", #gpu_runtime.region_desc<device =
"level_zero:gpu:0">> to !ntensor.ntensor<?x?xsi64 : "C">
    return %2 : !ntensor.ntensor<?x?xsi64 : "C">
  }
}
```

GPU pipeline: env region

- Need to preserve device info after linalg lowering
- **`EnvironmentRegionOp`**
  - Controls which parts of the code should be offloaded to specific device
  - Single block, can capture and yield values
  - Inhibits undesired optimizations
    - E.g., Prevents fusion between GPU and host ops
  - Have attached env attribute
  - Can be nested, innermost will take precedence usually
  - Various canonicalizations
    - Merge adjacent regions with same attribute – most important one

GPU pipeline: env region

```
func.func @foo(%arg0: memref<?x?xi32>) -> memref<?x?xi64, strided<[?, ?], offset: ?>> {
%c0 = arith.constant 0 : index
%c1 = arith.constant 1 : index
%0 = memref.get_global @__constant_xi64 : memref<i64>
%1 = numba_util.env_region #gpu_runtime.region_desc<device = "level_zero:gpu:0"> ->
memref<?x?xi64> {
  ...
  linalg.generic {indexing_maps = [#map, #map1, #map], iterator_types = ["parallel",
"parallel"]} ins(%arg0, %0 : memref<?x?xi32>, memref<i64>) outs(%alloc : memref<?x?xi64>) {
  ^bb0(%in: i32, %in_1: i64, %out: i64):
    ...
  }
  numba_util.env_region_yield %alloc : memref<?x?xi64>
}
...
}
```

GPU pipeline: scf.parallel

```
func.func @foo(%arg0: memref<?x?xi32>) -> memref<?x?xi64, strided<[?, ?], offset: ?>> {
...
%0 = numba_util.env_region #gpu_runtime.region_desc<device = "level_zero:gpu:0"> -> memref<?x?xi64> {
  %dim = memref.dim %arg0, %c0 : memref<?x?xi32>
  %dim_0 = memref.dim %arg0, %c1 : memref<?x?xi32>
  %alloc = memref.alloc(%dim, %dim_0) {alignment = 64 : i64} : memref<?x?xi64>
  scf.parallel (%arg1, %arg2) = (%c0, %c0) to (%dim, %dim_0) step (%c1, %c1) {
    %3 = memref.load %arg0[%arg1, %arg2] : memref<?x?xi32>
    %4 = arith.extsi %3 : i32 to i64
    %5 = arith.addi %4, %c1_i64 : i64
    memref.store %5, %alloc[%arg1, %arg2] : memref<?x?xi64>
    scf.yield
  }
  ...
  numba_util.env_region_yield %2 : memref<?x?xi64>
}
...
}
```

GPU pipeline: tile scf.parallel

```
%2:3 = gpu_runtime.suggest_block_size, %dim, %dim_0, %c1 -> index, index, index
%3 = arith.ceildivui %dim, %2#0 : index
%4 = arith.ceildivui %dim_0, %2#1 : index
scf.parallel (%arg1, %arg2, %arg3, %arg4, %arg5, %arg6) = (...) to (%3, %4, %c1, %2#0, %2#1, %c1) step (...)
{
%6 = arith.muli %arg1, %2#0 : index
%7 = arith.addi %6, %arg4 : index
%8 = arith.cmpi slt, %7, %dim : index
%9 = arith.muli %arg2, %2#1 : index
%10 = arith.addi %9, %arg5 : index
%11 = arith.cmpi slt, %10, %dim_0 : index
%12 = arith.andi %8, %11 : i1
scf.if %12 {
  %13 = memref.load %arg0[%7, %10] : memref<?x?xi32>
  %14 = arith.extsi %13 : i32 to i64
  %15 = arith.addi %14, %c1_i64 : i64
  memref.store %15, %alloc[%7, %10] : memref<?x?xi64>
}
scf.yield
} {mapping = [...]}
```

GPU pipeline: parallelLooptoGPU

```
%0 = numba_util.env_region #gpu_runtime.region_desc<device = "level_zero:gpu:0"> ->
memref<?x?xi64> {
  ...
  %2:3 = gpu_runtime.suggest_block_size, %dim, %dim_0, %c1 -> index, index, index
  %3 = arith.ceildivui %dim, %2#0 : index
  %4 = arith.ceildivui %dim_0, %2#1 : index
  gpu.launch blocks(%arg1, %arg2, %arg3) in (%arg7 = %3, %arg8 = %4, %arg9 = %c1)
threads(%arg4, %arg5, %arg6) in (%arg10 = %2#0, %arg11 = %2#1, %arg12 = %c1) {
    ...
    gpu.terminator
  }
  ...
  numba_util.env_region_yield %5 : memref<?x?xi64>
}
```

GPU pipeline: UnstrideMemrefsPass

```
%13 = memref.load %arg0[%7, %10] : memref<?x?xi32>
```

```
-- converted to --
```

```
%base_buffer, %offset, %sizes:2, %strides:2 = memref.extract_strided_metadata
%arg0 : memref<?x?xi32> -> memref<i32>, index, index, index, index, index
...
gpu.launch blocks
...
%13 = affine.apply #map()[%7, %strides#0, %10]
%reinterpret_cast = memref.reinterpret_cast %base_buffer to offset: [%13],
sizes: [], strides: [] : memref<i32> to memref<i32>
%14 = memref.load %reinterpret_cast[] : memref<i32>
```

GPU pipeline: Outline kernel

```
%0 = numba_util.env_region #gpu_runtime.region_desc<device =
"level_zero:gpu:0"> -> memref<?x?xi64> {
  ...
  %2:3 = gpu_runtime.suggest_block_size @foo_kernel, %dim, %dim_0, %c1
-> index, index, index
  %3 = arith.ceildivui %dim, %2#0 : index
  %4 = arith.ceildivui %dim_0, %2#1 : index
  gpu.launch_func  @foo_module::foo_kernel blocks in ...
  ...
  numba_util.env_region_yield %5 : memref<?x?xi64>
}
```

GPU pipeline: GPUExPass

```
%0 = gpu_runtime.create_gpu_stream, "level_zero:gpu:0"
%1 = numba_util.env_region #gpu_runtime.region_desc<device = "level_zero:gpu:0"> ->
memref<?x?xi64> {
  %3 = "gpu_runtime.load_gpu_module"(%0) {module = @foo_module}
  %4 = "gpu_runtime.get_gpu_kernel"(%3) {kernel = @foo_kernel}
  %5:3 = gpu_runtime.suggest_block_size : %0 %4, %dim, %dim_0, %c1 -> index, index, index
  %6 = arith.ceildivui %dim, %5#0 : index
  %7 = arith.ceildivui %dim_0, %5#1 : index
  "gpu_runtime.launch_gpu_kernel"(%0, %3, %6, %7, %c1, %5#0, %5#1, %c1) ...
  ...
  numba_util.env_region_yield %10 : memref<?x?xi64>
}
```

GPU pipeline: Init outlining 1

```
func.func private @foo_init() -> (!gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque) attributes
{plier.outlined_init} {
  %0 = gpu_runtime.create_gpu_stream, "level_zero:gpu:0"
  %1 = "gpu_runtime.load_gpu_module"(%0) {module = @foo_module} : (!gpu_runtime.stream) -> !gpu_runtime.opaque
  %2 = "gpu_runtime.get_gpu_kernel"(%1) {kernel = @foo_kernel} : (!gpu_runtime.opaque) -> !gpu_runtime.opaque
  return %0, %1, %2 : !gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque
}
func.func private @foo_deinit(%arg0: !gpu_runtime.stream, %arg1: !gpu_runtime.opaque, %arg2: !gpu_runtime.opaque) attributes
{plier.outlined_deinit} {
  "gpu_runtime.destroy_gpu_kernel"(%arg2) : (!gpu_runtime.opaque) -> ()
  "gpu_runtime.destroy_gpu_module"(%arg1) : (!gpu_runtime.opaque) -> ()
  gpu_runtime.destroy_gpu_stream %arg0
  return
}
func.func @foo(%arg0: memref<?x?xi32>) -> memref<?x?xi64, strided<[?, ?], offset: ?>> {
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %0:3 = call @foo_init() {plier.outlined_init} : () -> (!gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque)
  ...
  call @foo_deinit(%0#0, %0#1, %0#2) {plier.outlined_deinit} : (!gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque) -
> ()
  return %cast : memref<?x?xi64, strided<[?, ?], offset: ?>>
}
```

GPU pipeline: Init outlining 2

```
func.func private @foo_init() -> (!gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque) attributes
{plier.outlined_init} {
  %0 = gpu_runtime.create_gpu_stream, "level_zero:gpu:0"
  %1 = "gpu_runtime.load_gpu_module"(%0) {module = @foo_module} : (!gpu_runtime.stream) -> !gpu_runtime.opaque
  %2 = "gpu_runtime.get_gpu_kernel"(%1) {kernel = @foo_kernel} : (!gpu_runtime.opaque) -> !gpu_runtime.opaque
  return %0, %1, %2 : !gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque
}
func.func private @foo_deinit(%arg0: !gpu_runtime.stream, %arg1: !gpu_runtime.opaque, %arg2: !gpu_runtime.opaque) attributes
{plier.outlined_deinit} {
  "gpu_runtime.destroy_gpu_kernel"(%arg2) : (!gpu_runtime.opaque) -> ()
  "gpu_runtime.destroy_gpu_module"(%arg1) : (!gpu_runtime.opaque) -> ()
  gpu_runtime.destroy_gpu_stream %arg0
  return
}
func.func @foo(%arg0: memref<?x?xi32>) -> memref<?x?xi64, strided<[?, ?], offset: ?>> {
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %context, %results:3 = "numba_util.take_context"() {initFunc = @foo_init, releaseFunc = @foo_deinit} : () ->
(!numba_util.opaque, !gpu_runtime.stream, !gpu_runtime.opaque, !gpu_runtime.opaque)
  ...
  "numba_util.release_context"(%context) : (!numba_util.opaque) -> ()
  return %cast : memref<?x?xi64, strided<[?, ?], offset: ?>>
}
```

FP64 emulation/Truncation for GPU

- Some devices doesn’t support fp64
- Truncate all arith ops inside kernel to fp32
- Reinterpret input memrefs as vector<2xi32>
- Use bitwise ops to manually convert to fp32
- Needs **`memref.bitcast`** op

Other

CFG-to-SCF

- Convert arbitrary CFG to SCF (scf.if's and scf.while's)
- Based on paper https://dl.acm.org/doi/pdf/10.1145/2693261
  - Loop restructuring
  - Branch restructuring
- scf.while later uplifted to scf.for and the to scf.parallel

Refcounted memrefs

- Need to interoperability with numba/numpy arrays
- (ab)use “allocated” pointer points for control block
- memref.clone -> util.retain
  - util.retain is view like and just increaments refcount
- Can attach arbitrary data to control block
  - Special flag for global memrefs to ignore refcount
  - SYCL queue for ComputeFlollowsData
- Simplifies ABI
  - Caller responsibility to deallocate returned me mrefs

Tuple support

- Supported as Ntensor numpy ops arguments
- Tuple type from upstream
- `BuildTupleOp/TupleExtractOp`

Signeddness

- Signed/Unsigned types are preserved on early stages
- **`SignCastOp`**

Loop fusion on scf.parallel

- naivelyFuseParallelOps
- Use LocalAliasAnalysis

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

NTensor dialect

- Dialect for high-level Numpy types and ops
- Types:
  - NTensorType – main array type
- High level ops:
  - Call
  - Unary
  - Binary
  - Get/SetItem
- Low level ops:
  - Load/Store/Dim/Subview
  - Copy
  - Elementwise
  - Broadcast
  - Primitive
- Conversion ops:
  - From/To/Tensor/Memref