PyDSL

A Python Subset for a better MLIR programming experience: Updates on new features and development

James Cox jellyfish3.14 jamesthejellyfish

Ritsuka ritsuka314 Ritsuka314

Kevin Lee cacophonia KevinLeeFM

Fabien Zhao Fabien fabienzz

Kai-Ting Amy Wang kai.ting.wang@huawei.com

Background: What is PyDSL?

- Python-based language that compiles to MLIR.
- @compile decorator turns a Python function into PyDSL and automatically compiles it.
- Pipeline: parses the AST, converts it to MLIR, lowers to shared library binary file.
- Compiled PyDSL functions can be called directly from Python. Arguments are converted to **ctype**, passed to shared library file.
- Has multiple supported backends including CPU and NPU. Can also be passed through existing MLIR pipelines.

```
import numpy as np

from pydsl.frontend import compile
from pydsl.memref import MemRef
from pydsl.scf import range
from pydsl.type import SInt16, SInt32

@compile()
def f(arr: MemRef[SInt32, 8, 4], x: SInt32, y: SInt16) ->
SInt32:
    z = x + y
    for i in range(8):
        for j in range(4):
            arr[i, j] += z

    return z

arr = np.zeros((8, 4), dtype=np.int32)
sm = f(arr, 7, 9)
print(arr)
print(sm)
```

Template Support

- Templates support PyDSL types as well as Python literals.
- Replaces the @compile() decorator, adds template descriptors at runtime to the AST's type parameters, and then everything is handled by the standard PyDSL compilation pipeline.
- @template() defers compilation until the function is called, creating a binary for each set of descriptors.
- Similar in design to C++ templates.
- Function caching is still working in progress.

adapted from tests/e2e/test_template.py

```
@template()
def calc[T, N, M](mat: Tensor[T, N, M]) -> Tensor[T, N, M]:
    n = Index(N)
    m = Index(M)
    mat[n - 1, m - 1] = 1
    return mat

arr = calc[F32, 10, 5](np.zeros((10, 5), dtype=np.float32))
arr = calc[F64, 1, 40](np.zeros((1, 40), dtype=np.float64))
```

Autotuning support

- PyDSL has autotune support for parameters such as tile sizes, types, and anything that is a variable within a kernel.
- Using the @autotune decorator, a user can specify any number of autotune parameters.
- Performs an exhaustive search, compiling each possibility. Therefore it is best for small configuration spaces.

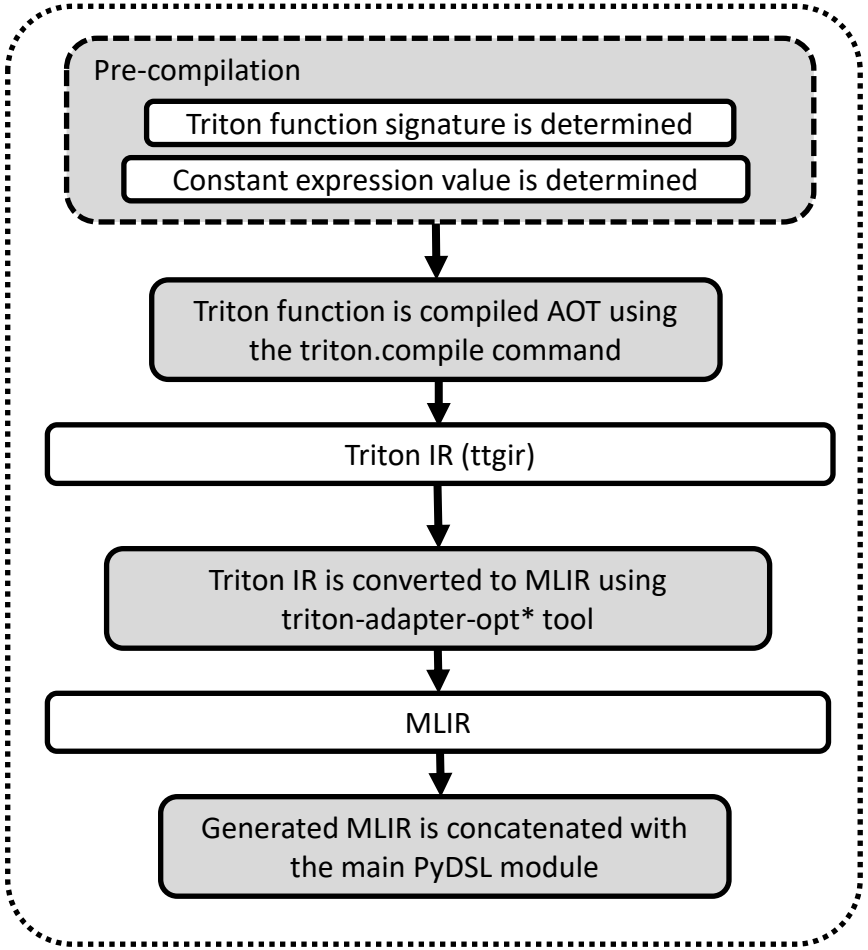
adapted from examples/autotune/test_program.py

```
autotune_values = Var("NUM", [100, 200, 600]) *
                                TestingData([[N,A]])


@autotune(autotune_values)
def heat_fuse_tile(n: Index, A: MemRef[F32, N, N]):
    v = Index(NUM)
    for _ in arange(S(v)):
        for i in arange(n):
            for j in arange(n):
                # -3/2 x^2 + 5/2 x + 1
                A[i, j] = (-1.5) * A[i, j] * A[i, j] + (2.5)
                              * A[i, j] + 1
```

Inter-operability with Triton

- Triton JITFunction kernels can be directly called by PyDSL functions.
- Currently PyDSL handles tiling, triton kernel is executed as a single thread
- Multi-core interop support is planned for NPU target



from examples/triton_interop.py:

```
@triton.jit
def kernel(x_ptr, y_ptr, output_ptr, n_elements,
BLOCK_SIZE):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0,
                          BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets,
             mask=mask)
    y = tl.load(y_ptr + offsets,
             mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output,
             mask=mask)
```

```
@compile()
def func_test(x: ArrayType, y: ArrayType, out:
ArrayType):
    size: Index = 98432
    BLOCK_SIZE = 64
    index_size = size // BLOCK_SIZE
    for i in arange(0, index_size):
        kernel(x, y, out, size,
             BLOCK_SIZE, i)
```

*triton-adapter-opt available in open-source repo https://gitcode.com/Ascend/triton-ascend

Improvements Made during the Past Year

1. **Memref/Tensor indexing. Uses Python's slicing syntax to implement memref.subview, tensor.extract_slice, tensor.insert_slice**

2. **Support Memref with strided layout (not shown)**

3. **Powerful reduce op**
   1. Reduce along multiple dimensions
   2. Specify custom combiner function using inline functions

4. **CallMacros that make the language extensible**
   1. Python functions that evaluate at compile time
   2. Python AST -> MLIR
   3. Most part of PyDSL is implemented via CallMacros
   4. Used to implement Ascend A3 target

1.
```
t1[()] = 456    # t1: Tensor[UInt32]
t2[1] = 5       # t2: Tensor[UInt32, 2]
t2x2[1, 1] = 5  # t2x2: Tensor[UInt32, 2, 2]

t1[2:9:3, 3:6] = t1[4:7, 1:7:2]
```

3.
```
@InlineFunction.generate()
def sum(a: UInt64, b: UInt32) -> UInt64:
    return a + b

@compile()
def f(
    arr: MemRef[UInt32, DYNAMIC, DYNAMIC, DYNAMIC],
    out: MemRef[UInt64, DYNAMIC],
):
    linalg.reduce(sum, arr, init=out, dims=[0, 2])
```

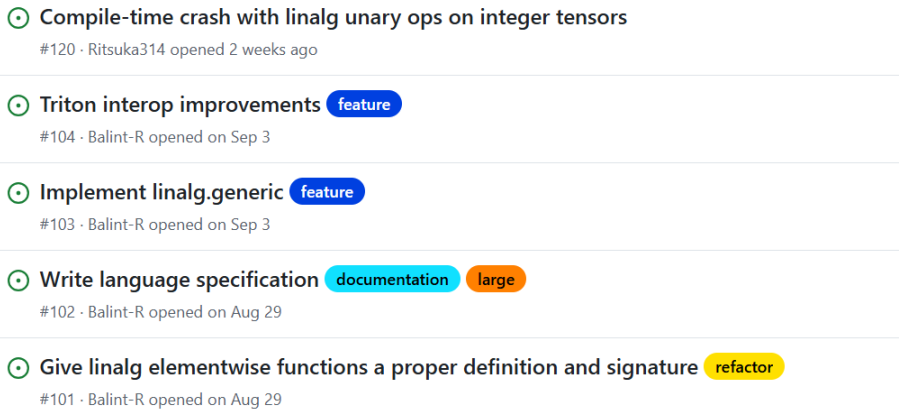
4.
```
MemRefF32GM = MemRef.get((16, 8), F32, memory_space=NPU_AddrSpace.GM)
@compile(target_class=AscendTarget)
def f(in: MemRefF32GM, out: MemRefF32GM):
  m = alloc((16, 8), F32, memory_space=NPU_AddrSpace.UB)
  npu.load(in, m)
  npu.vexp(m, out=m)
  npu.store(m, out)

@CallMacro.generate()
def store(visitor: ToMLIRBase, src: Compiled, dst: Compiled):
    # ...  Type inferencing
    rep = npu.StoreOp(result_tensor_type, lower_single(src), lower_single(dst))
```

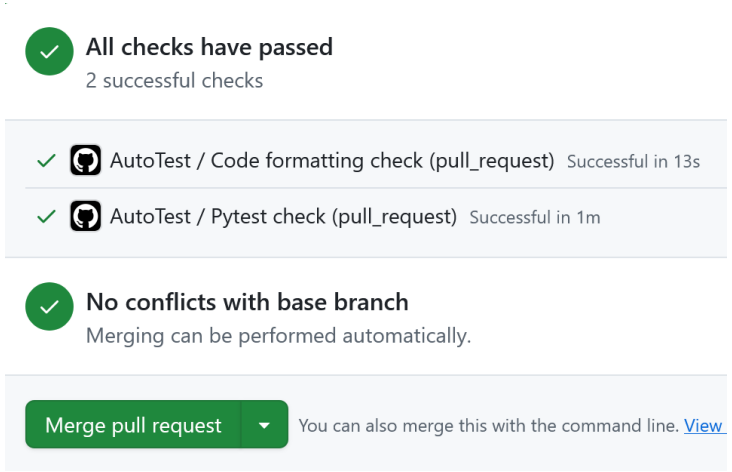
Open source development

- Everything is open-sourced on GitHub (https://github.com/Huawei-CPLLab/PyDSL), with integrated CI and active development
- Everyone is free to contribute! Many issues raised to show proposed features/changes
- Language guide: PyDSL/blob/main/docs/usage.md
- Any questions regarding PyDSL can be asked on the discord:
- https://tinyurl.com/PyDSLdiscord

Issues



CI



Pull requests