2026 US LLVM Developers' Meeting

27 October 2026

MLIR DSL: An Untraditional DSL for an Untraditional Compiler

How to build a Python DSL for MLIR

Guray Ozen

Principal Compiler Engineer, NVIDIA

Amir Mohammad Tavakkoli

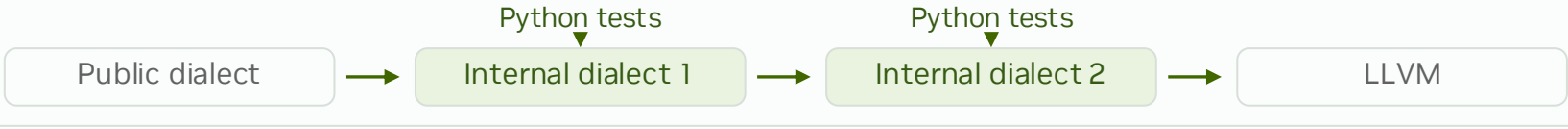
PhD student, University of Utah

NVIDIA

Where do we need to produce MLIR?

Four internal use cases I discovered.

1 – Test internal dialects



2 – Generate recipe IR



3 – Reuse DSL building blocks



Your DSL

JIT | Cache | Env

4 – Build a full language



CuTeDSL

CUTLASS Python

Let’s compute a + b with Python bindings

Verbose code / boilerplate 

The issue: boilerplate for a + b

1 · Build IR

```
import ctypes as C
from mlir import ir, execution_engine as ee
from mlir import passmanager as pm
from mlir.dialects import arith, func

with ir.Context(), ir.Location.unknown():
    i32 = ir.IntegerType.get_signless(32)
    module = ir.Module.create()
    with ir.InsertionPoint(module.body):
        fn = func.FuncOp(
            "add", ([i32, i32], [i32]))
        attrs = fn.attributes
        attrs["llvm.emit_c_interface"] = (
            ir.UnitAttr.get())
        entry = fn.add_entry_block()
        with ir.InsertionPoint(entry):
            a, b = fn.arguments
            sum_op = arith.AddIOp(a, b)
            func.ReturnOp([sum_op.result])
```

2 · Compile + call

```
pipeline = (
    "builtin.module("
    "convert-arith-to-llvm,"
    "convert-func-to-llvm,"
    "reconcile-unrealized-casts)"
)
pm.PassManager.parse(pipeline).run(
    module.operation)
engine = ee.ExecutionEngine(module)

a, b = C.c_int32(2), C.c_int32(3)
out = C.c_int32()
engine.invoke(
    "add", C.byref(a), C.byref(b),
    C.byref(out))
print(out.value)  # 5
```

What else must we build?

PyTorch / NumPy inputs?

Ahead-of-time compilation?

JIT cache?

Control flow?

The answer: `mlir dsl`

Write the computation. Reuse the infrastructure.

Raw Python bindings · complete program



1 · Build IR

```
import ctypes as C
from mlir import ir, execution_engine as ee
from mlir import passmanager as pm
from mlir.dialects import arith, func

with ir.Context(), ir.Location.unknown():
    i32 = ir.IntegerType.get_signless(32)
    module = ir.Module.create()
    with ir.InsertionPoint(module.body):
        fn = func.FuncOp(
            "add", ([i32, i32], [i32]))
        attrs = fn.attributes
        attrs["llvm.emit_c_interface"] = (
            ir.UnitAttr.get())
        entry = fn.add_entry_block()
        with ir.InsertionPoint(entry):
            a, b = fn.arguments
            sum_op = arith.AddIOp(a, b)
            func.ReturnOp([sum_op.result])
```

2 · Compile + call

```
pipeline = (
    "builtin.module("
    "convert-arith-to-llvm,"
    "convert-func-to-llvm,"
    "reconcile-unrealized-casts)"
)
pm.PassManager.parse(pipeline).run(
    module.operation)
engine = ee.ExecutionEngine(module)

a, b = C.c_int32(2), C.c_int32(3)
out = C.c_int32()
engine.invoke(
    "add", C.byref(a), C.byref(b),
    C.byref(out))
print(out.value)  # 5
```

This is what you want!



`mlir dsl`

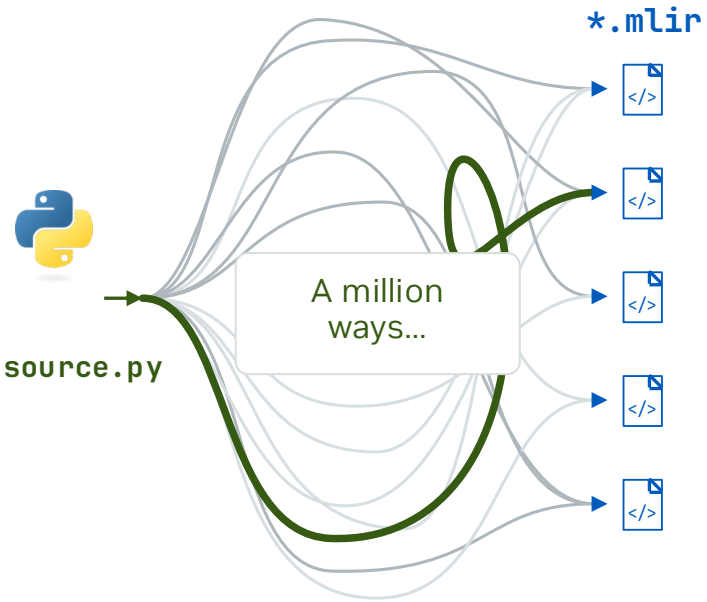
```
import mlir.mlir_dsl as m

@m.jit
def add(a: m.Int32,
        b: m.Int32):
    return a + b

print(add(2, 3))
```

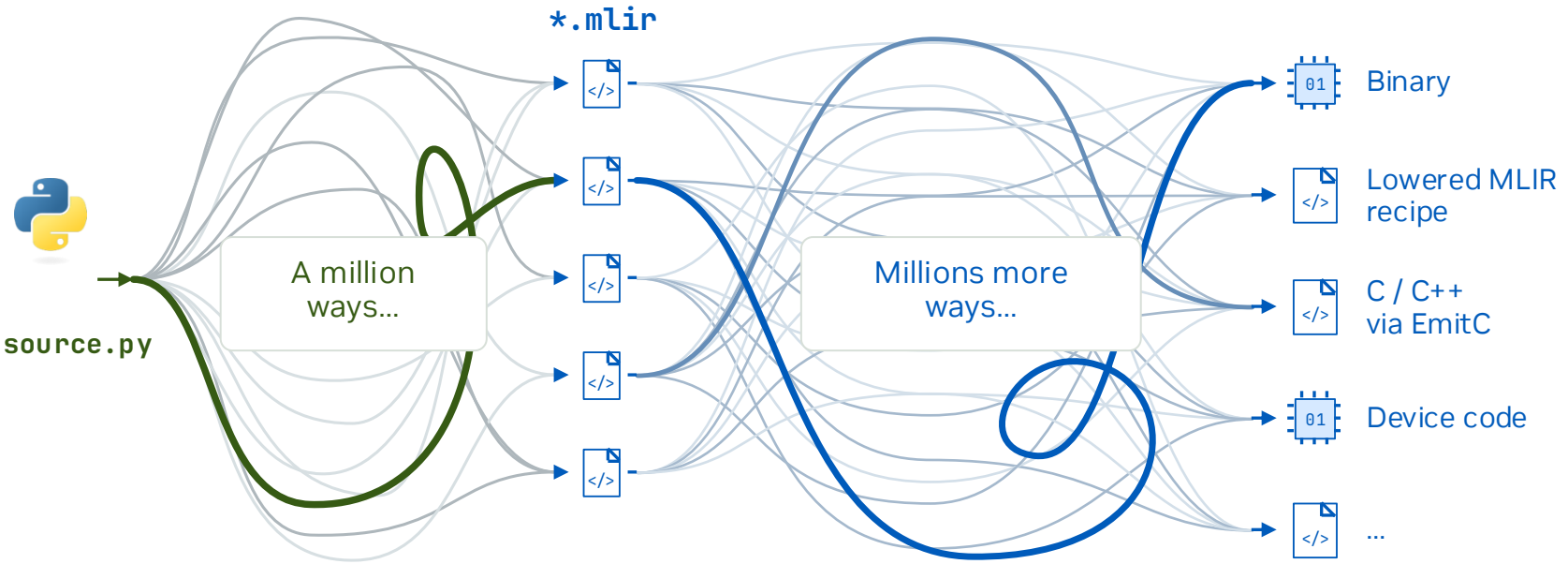
MLIR isn’t a traditional compiler

Python generates MLIR; each compiler chooses its path from there.



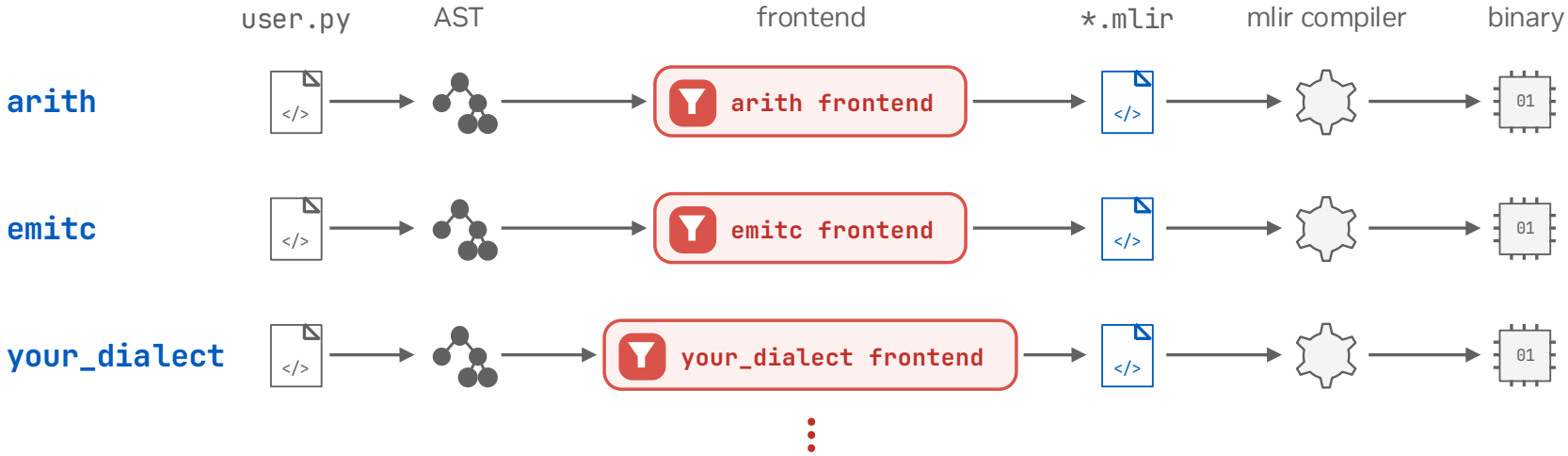
MLIR isn't a traditional compiler

Python generates MLIR; each compiler chooses its path from there.



Can we build a traditional frontend for MLIR? No.

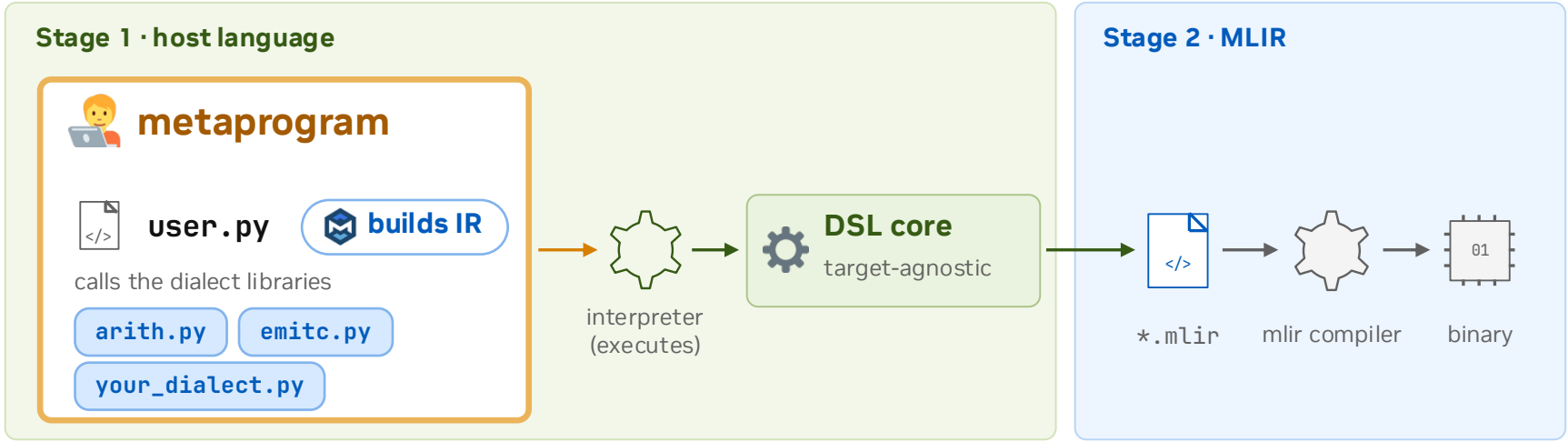
A traditional compiler needs a frontend for every dialect.



✗ **Every new dialect needs its own frontend: arith, emitc, yours … it does not scale.**

Build the IR on the fly: the user shapes it, not the DSL

Metaprogramming is multi-stage programming[1]: building the IR moves to the user's side.



✓ **No frontend per dialect: the metaprogram calls the dialect libraries and builds the IR.**

[1] W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. PEPM '97, pp. 203–217.

Which choices define your DSL?

Five plugin types. Select what your DSL needs.

MLIR is diverse. How do we adapt?		Many possible answers
1	Types + operators Which target dialect?	→ `arith`? `emitc`? `your_arith`? what?
2	Control flow Which capture and region rules?	→ `scf`? your regions? none? what?
3	Compilation Which backend, or IR only?	→ `JIT`? IR only? your backend? what?
4	Function entry Which decorators and call rules?	→ `@jit`? `@kernel`? your entry? what?
5	Framework memory How do arrays and tensors get in?	→ `numpy`? `torch`? `jax`? `memref`? what?

Which choices define your DSL?

Proposal: plugins. Five plugin types: select what your DSL needs.

MLIR is diverse. How do we adapt?

1	Types + operators	Which target dialect?
2	Control flow	Which capture and region rules?
3	Compilation	Which backend, or IR only?
4	Function entry	Which decorators and call rules?
5	Framework memory	How do arrays and tensors get in?

Five plugin types

1	Types + operations	`arith` or your dialect
2	AST preprocessor	`scf`, `custom`, or `none`
3	Compiler / JIT	`ExecutionEngine` or your backend
4	Decorators	`@jit`, `@kernel`, or your entry
5	Adapters	`NumPy` · `PyTorch` · `DLPack`

Compose your DSL from the shared core

BaseDSL is shared infrastructure, not an end-to-end compiler.

BaseDSL

Dialect-independent core

Tracing

Type inference

Diagnostics

JIT cache

inherit

`class YourDSL(BaseDSL):`

Five plugin types

Roles · zero or one each

1. Types + operations
2. AST preprocessor
3. Compiler / JIT

Families · any number

4. Decorators
5. Adapters

Your pass pipeline `pipeline()`

No compiler → generate IR

Compiler + pipeline → execute

What we implemented

BaseDSL is the shared infrastructure. The plugins are what we wrote to make it work for this tutorial.

BaseDSL

10,367 lines of code in 16 files

Tracing and the compile pipeline	1.8k
Staging and the host boundary	1.5k
The type system · 30 dtypes	3.3k
The plugin machinery	0.6k
The JIT cache · memory and disk	0.3k
Diagnostics · 41 error codes	1.9k
Settings, logging, profiler	1.0k

Plugins

8 plugins written for the tutorial

Types	2	arith · vector · llvm, emitc
AST	1	scf
Compiler	1	MLIR JIT
Decorators	2	@jit, @kernel
Adapters	2	DLPack, TVM-FFI

2 test DSLs, MLIRTestDSL and EmitCTestDSL · 14 examples · 35 lit tests

The Tensor IR and Triton DSLs bring plugins of their own.

Four DSLs on one shared core

Implemented for this talk: one BaseDSL core, four Plugins records, four different outputs.

DSL	Plugins it names: Types	AST	Compiler	Decorators	Adapters	Generates
1 **`MLIRTestDSL(BaseDSL)`** the reference DSL, in the tree	arith	scf	MLIR JIT	@jit, @kernel	DLPack, TVM-FFI	`arith · scf · func` + vector, llvm, gpu
2 **`EmitCTestDSL(BaseDSL)`** the same Python, emitc	emitc	scf	–	@jit	–	`emitc` then C++ (mlir-translate)
3 **`TensorIRDSL(BaseDSL)`** NVIDIA's Tensor IR (prototype)	tensor_ir	–	CudaTile	@kernel	DlpackTensors	`nv_tensor_ir` a graph of tensor ops
4 **`TritonDSL(BaseDSL)`** OpenAI's Triton DSL (PR)	arith	Triton	Triton	@kernel	TritonTensors, Closures	`tt` Triton IR, stock Triton

Better than Triton's current frontend: it enables metaprogramming.

@triton.jit reads the kernel's Python. The DSL runs it, so plain functions, closures and classes can build the kernel.

Agenda

01 Decorator plugin

02 Adapter plugin and entry generator

03 Type inference

04 Language virtualization

05 Compile, run and diagnostics

06 Mutation tracking mechanism

advanced · story from the CUTLASS Python DSL

07 Examples

1. MLIRTest DSL
2. EmitC DSL
3. NVIDIA's TensorIR DSL
4. OpenAI's Triton DSL

08 Conclusion

Concept: The lifecycle of one DSL call

Step 1 only wraps. A @jit call runs steps 2–8: interpret (trace), JIT on a memory miss.

1. **Decorate**
   Return the call wrapper.
   DecoratorPlugin · name
2. **Call wrapper**
   Install plugins; read settings.
   BaseDSL · plugin setup
3. **AST rewrite**
   Insert optional callbacks.
   ASTPreprocessorPlugin
4. **Dispatch**
   Select host or nested call.
   DecoratorPlugin · call or launch
5. **Adapt inputs**
   Bind, adapt and marshal inputs.
   AdapterPlugin · host objects
6. **Entry generator**
   Type the arguments, build the entry.
   BaseDSL · the decorator plugin builds the op
7. **Trace Python**
   Execute Python; finalize MLIR.
   TypeOpsPlugin + AST callbacks
8. **Build / invoke**
   Cache or compile, then invoke.
   CompilerPlugin · compile, JIT, load, invoke

Example: The lifecycle on our program

The eight steps of the previous slide on one program: each disc marks where a step works.

Python · user code

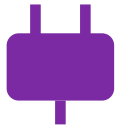
```
@m.jit                                          1
def saxpy(a: m.Float32, x, y,                   5 6
          n: m.Int32):
    for i in m.range(n):                        3 7
        y[i] = a * x[i] + y[i]                    7

x = torch.arange(8, dtype=torch.float32)
y = torch.ones(8)
saxpy(2.0, x, y, 8)                             2 4 8
```

BaseDSL's own code, not a plugin: it asks each value's class for its MLIR type, and the decorator plugin builds the entry op. Extend it by registering classes.

Who works at each step

1 →	**Decorator plugin**	wraps the function
2 →	**BaseDSL**	installs the plugins
3 →	**AST plugin**	rewrites for and if
4 →	**Decorator plugin**	dispatches the call
5 →	**Adapter plugin**	tensor → DSL value
6 →	**Entry generator**	types the %args
7 →	**Types plugin**	emits the operations
8 →	**Compiler plugin**	lowers, JITs, runs



A decorator plugin

Concept: The decorator plugin

A DSL names its decorators. Each decorator is a plugin that does three jobs.

❶ Name it	→ ❹ Dispatch	→ ❻ Build the entry op
`@m.jit` `def host(...)`	`host(...)`	`func.func @host`
`@m.kernel` `def scale(...)`	`scale(...).launch(...)`	`gpu.func @kernel_scale_0`
Decorating only wraps the function: no body runs and nothing compiles.	From plain Python, @jit runs the function; in a trace, @kernel prepares a launch.	The operation the body is traced into: func.func for @jit, gpu.func for @kernel, or your own.

Each decorator is a plugin of its own: @m.jit and @m.kernel are two of them.

Example: Add a decorator

You implement a plugin that answers the calls and builds the entry. The DSL class plugs it in.

Plugin · a decorator, named in the record

```
# gpu_plugin.py
class KernelsPlugin(DecoratorPlugin):
    # the target-neutral half
    decorator_name = "kernel"
    # a call inside a trace: only prepare it
    def launch(self, dsl, func, *args, **kwargs):
        return KernelLauncher(dsl, self, func,
                              args, kwargs)

class Kernels(KernelsPlugin):
    # the gpu half: the entry is a gpu.func
    def generate_func_op(self, name, arg_types,
                         arg_attrs, loc=None):
        ...
```

plug it in

DSL · the class that plugs it in

```
class MLIRTestDSL(BaseDSL):
    plugins = Plugins(
        ...,
        decorators=[
            func.Jit(),
            gpu_plugin.Kernels(),
        ],
    )
```

The decorators of MLIRTestDSL

`@m.jit`	`from func.Jit()`
`@m.kernel`	`from gpu_plugin.Kernels()`

gpu_plugin.Kernels() is the class Kernels in gpu_plugin.py, and it extends KernelsPlugin.

Example: Use the decorators

One question decides which hook runs: is a trace already open?

Python · user code

```
@m.kernel
def scale(x: m.Pointer[m.Float32], a: m.Float32):
    x[0] = a * x[0]

@m.jit
def host(x: m.Pointer[m.Float32], a: m.Float32):
    scale(x, a).launch(grid=[1, 1, 1],
                       block=[1, 1, 1])

host(x, 2.0)
```

Is a trace already open?

No: a call from plain Python

`host(x, 2.0)` → `call` → trace, JIT, run

```
func.func @host(%arg0: !llvm.ptr, %arg1: f32) { … }
```

Yes: inside the trace of host

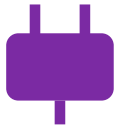
`scale(x, a)` → `launch` → prepare the launch

no IR yet

`.launch(...)` → `emit` → issue the launch

```
gpu.func @kernel_scale_0(…) kernel { … }
gpu.launch_func @kernels::@kernel_scale_0 …
```

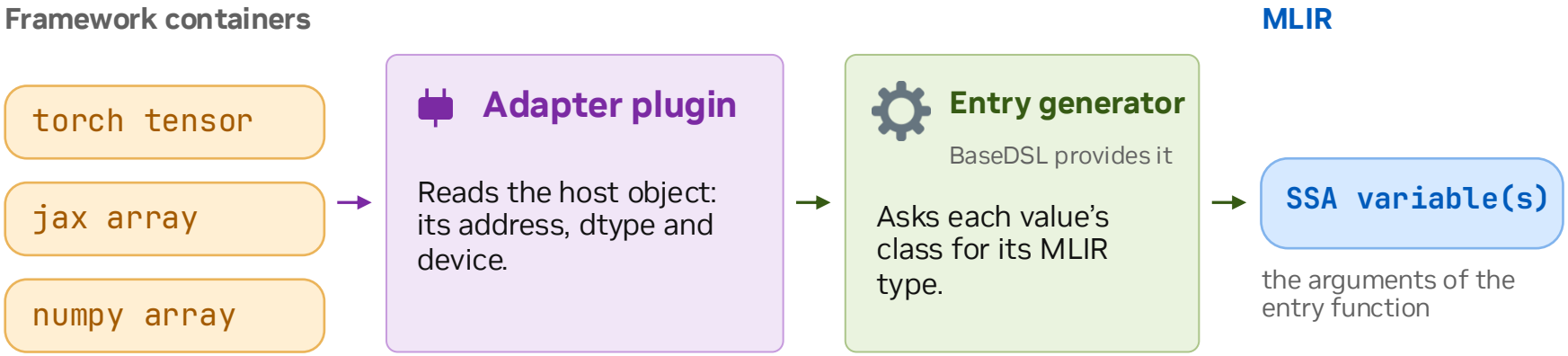
A @jit call from a trace inlines its body. A @kernel call only prepares the launch, and .launch issues it.



Adapter plugin

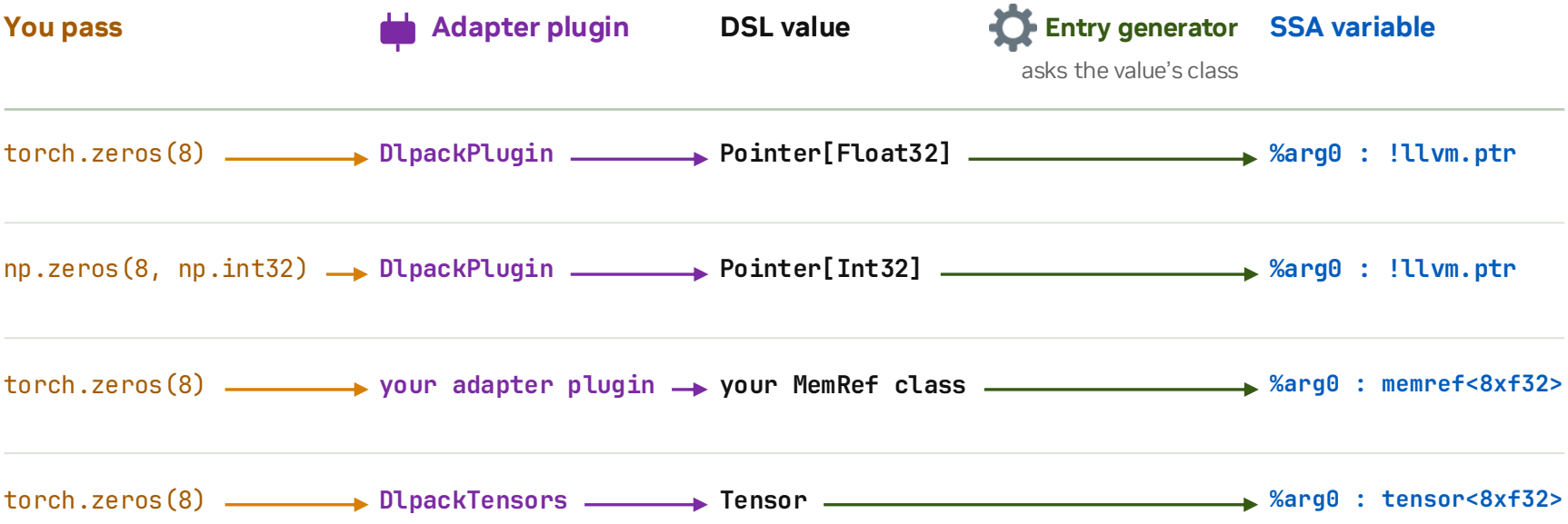
Concept: The adapter plugin

A framework tensor is not an MLIR value. The adapter plugin and the entry generator make it one.



Example: One tensor, different plugins

The adapter plugin makes the DSL value. BaseDSL's entry generator asks that value's class for its MLIR type.





The entry generator

(BaseDSL provides)

Concept: The entry generator

It gives each entry argument its MLIR type. Three things decide it: the call, the annotation, the generic annotation.

The call passes	`@m.jit` `def f(a)` 1 · the value decides	`@m.jit` `def f(a: m.Int32)` 2 · the annotation decides	`@m.jit` `def f(a: m.Numeric)` 3 · the generic annotation
`m.Int8(1)`	`i8`	`i32`	`i8`
`m.Int16(0)`	`i16`	`i32`	`i16`
`m.Int32(0)`	`i32`	`i32`	`i32`
`m.Int64(1)`	`i64`	`i32`	`i64`
`m.Float16(1.0)`	`f16`	`i32`	`f16`
`m.Float32(1.0)`	`f32`	`i32`	`f32`
`0`	no MLIR value, metaprogram value	`i32`	`i32`
`2.5`	no MLIR value, metaprogram value	`i32`	`f32`
`True`	no MLIR value, metaprogram value	`i32`	`i1`

Example: A type comes from the annotation or the value

Duck typing: the same function is f32 or i32 by the values it is given.

Python · no annotations



```python
@m.jit
def axpy(a, x, y):
    return a * x + y

F, I = m.Float32, m.Int32

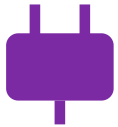
# DSL values: f32 parameters
axpy(F(2.0), F(3.0), F(4.0))

# DSL values: i32 parameters
axpy(I(2), I(3), I(4))

# Python values: metaprogram
axpy(2, 3, 4)
```

Generated MLIR · one module per call

```
// axpy(F(2.0), F(3.0), F(4.0))
func.func @axpy(%arg0: f32, %arg1: f32, %arg2: f32) -> f32
    attributes {llvm.emit_c_interface} {
  %0 = arith.mulf %arg0, %arg1 : f32
  %1 = arith.addf %0, %arg2 : f32
  return %1 : f32
}
// axpy(I(2), I(3), I(4))
func.func @axpy(%arg0: i32, %arg1: i32, %arg2: i32) -> i32
    attributes {llvm.emit_c_interface} {
  %0 = arith.muli %arg0, %arg1 : i32
  %1 = arith.addi %0, %arg2 : i32
  return %1 : i32
}
// axpy(2, 3, 4)
func.func @axpy_2_3_4() -> i32 attributes {llvm.emit_c_interface} {
  %c10_i32 = arith.constant 10 : i32
  return %c10_i32 : i32
}
```



Types plugin

Concept: A DSL value is a Numeric

A Numeric holds one value, and its dtype is its class. The core ships a class for every dtype.

DSL · class Numeric, in the core

```
class Numeric:
    __slots__ = ("value",)
    @property
    def dtype(self):
        return type(self)
```

Integer	`Boolean` `Int2` `Int4` `Int8` `Int16` `Int32` `Int64` `Int128`
Unsigned	`Uint8` `Uint16` `Uint32` `Uint64` `Uint128`
Float	`Float16` `BFloat16` `TFloat32` `Float32` `Float64`
8-bit float	`Float8E4M3FN` `Float8E5M2` `Float8E4M3` `Float8E3M4`
	`Float8E8M0FNU` `Float8E5M3FNU` `Float8E5M2FNUZ` `Float8E4M3FNUZ`
and more	`Float8E4M3B11FNUZ` `Float6E3M2FN` `Float6E2M3FN` `Float4E2M1FN`

30 dtypes in m.*. Each class names its MLIR type: Int8 → i8, Float16 → f16, Float8E4M3FN → f8E4M3FN …

Also in the core `Vector` N lanes of one Numeric dtype `Pointer` an address of a Numeric dtype

Not Numerics, and target-agnostic: the types plugin gives each its MLIR type and operations.

Concept: The Numeric class defines the rules

Promotion and casts are the Numeric class's. What an operator generates is the types plugin's.

The Numeric class defines the rules



```
Numeric   op   Numeric

Numeric   op   Python value
```

Promotion and casts are the core's: one set of rules for every DSL.

asks →

Types plugin

You plug what Numeric generates:

- `arith`
- `emitc`
- `tensor_ir`
- `your_dialect`

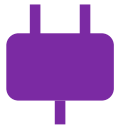
Other rules need a value class of their own: the Tensor IR DSL's tensors take the same dtypes only.

Concept: One expression, three types plugins

The Numeric class picks the dtype. Each types plugin generates its own operations.

You write	type a	type b	arith MLIRTestDSL	emitc EmitCTestDSL	tensor_ir TensorIRDSL · tensors
`p = a * x`	Float32	Float32	`arith.mulf`	`emitc.mul`	`nv_tensor_ir.mul`
`p = a * x`	Float16	Float32	`arith.extf · arith.mulf`	`arith.extf · emitc.mul`	✗ same dtypes only
`q = n * x`	Int32	Float32	`arith.sitofp · arith.mulf`	`arith.sitofp · emitc.mul`	✗ same dtypes only
`k + 1`	Int8	Python int	`arith.extsi · arith.addi`	`arith.extsi · emitc.add`	✗ tensors only
`n / d`	Int32	Int32	`arith.sitofp ×2 · arith.divf`	same	`nv_tensor_ir.div`
`a < b`	Int32	Int32	`arith.cmpi`	same	—
`r.to(m.Int32)`	Float32	→ Int32	`arith.fptosi`	same	—

Constants are not shown. Only add and mul are EmitC; the rest stays arith.



AST-preprocessor plugin

Example: Plain Python is metaprogramming

A trace runs your Python. if, for and assert on Python values run while the IR is built.

Python · user code



```
@m.jit
def k(x, y, t, n: m.Int32):
    assert t > 0
    for i in range(4):
        if i < t:
            y[i] = x[i] + 1
```

Traced: Python runs it

→ **checked once, while tracing**
→ **unrolled: the body is traced four times**
→ **decided each step: only taken branches stay**

Do you want to virtualize the Python?

Example: Virtualize with the AST plugin

m.assert_, m.range and m.expr mark what the DSL captures; the AST plugin rewrites it into callbacks.

Python · user code

```
@m.jit
def k(x, y, t: m.Int32, n: m.Int32):
    # Virtualize assert
    m.assert_(t > 0)

    # Virtualize the for loop
    for i in m.range(n):

        # Virtualize the if statement
        if m.expr(i < t):
            y[i] = x[i] + 1
```

m.range exists; m.assert_ and m.expr are the spellings we propose.



AST plugin

Plugin · generated Python

```
@m.jit
def k(x, y, t: m.Int32, n: m.Int32):
    # Virtualize assert
    m.assert_executor(t > 0)
    # Virtualize the for loop
    @m.loop_selector(0, n, 1)
    def loop_body(i, y):
        # Virtualize the if statement
        @m.if_selector(i < t)
        def if_region(pred, y):
            def then_block(y):
                y[i] = x[i] + 1
                return [y]
            return if_executor(pred, then_block)
        y = if_region
        return [y]
    y = loop_body
```

Example: One body, two loops

m.range is captured and builds scf.for; a plain range over a Python value runs in Python and unrolls.

Python · user code



```python
@m.jit
def staged(x: m.Int32, n: m.Int32):
    s = x
    for i in m.range(n):
        s += x
    return s

@m.jit
def unrolled(x: m.Int32):
    s = x
    for i in range(3):
        s += x
    return s
```

Generated MLIR

```
func.func @staged(%arg0: i32, %arg1: i32) -> i32 {
  %c0_i32 = arith.constant 0 : i32
  %c1_i32 = arith.constant 1 : i32
  %0 = scf.for %arg2 = %c0_i32 to %arg1 step %c1_i32
      iter_args(%arg3 = %arg0) -> (i32)  : i32 {
    %1 = arith.addi %arg3, %arg0 : i32
    scf.yield %1 : i32
  }
  return %0 : i32
}

func.func @unrolled(%arg0: i32) -> i32 {
  %0 = arith.addi %arg0, %arg0 : i32
  %1 = arith.addi %0, %arg0 : i32
  %2 = arith.addi %1, %arg0 : i32
  return %2 : i32
}
```

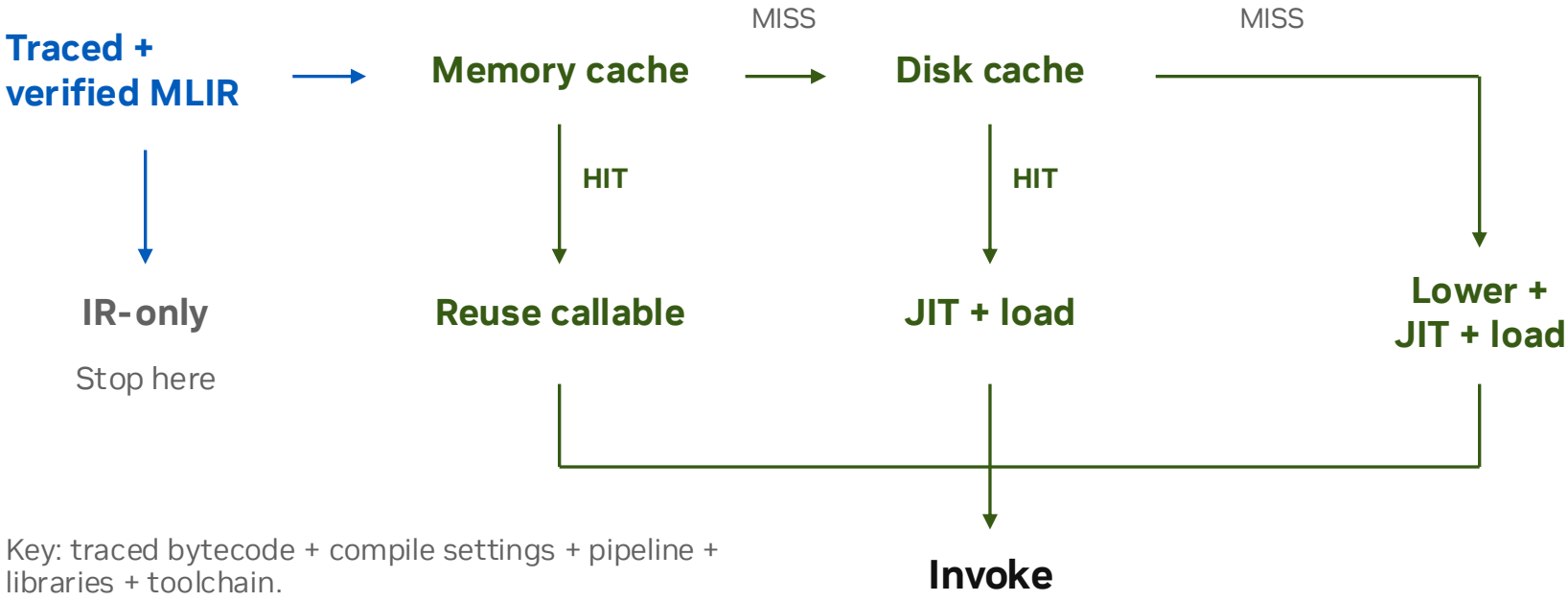


JIT Cache

(provided by BaseDSL)

Concept: Hash the module, then look up the caches

Every ordinary call reaches this point after tracing; only a memory miss is JIT-compiled.



Key: traced bytecode + compile settings + pipeline + libraries + toolchain.

Concept: The cache algorithm

Two levels, one key: a hit skips the work behind it.

DSL · the lookup, in short

```
key = get_module_hash(module, pipeline)        # the SHA-256 key
fn = jit_cache.get(key)                        # memory lookup
if fn is None:                                 # memory miss
    lowered = load_cache_from_path(key)        # disk lookup
    if lowered:                                # file found
        engine = jit_lowered_module(lowered)   # skip the passes
    else:                                      # no file
        engine = compile_and_jit(module)       # passes + JIT
    fn = load(engine)                          # the callable
    jit_cache.set(key, fn)                     # keep in memory
    if not lowered:                            # built just now
        dump_cache_to_path(key, fn)            # save to disk
return fn(args)                                # run it
```

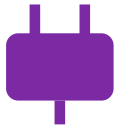
The key is a SHA-256 of

- the traced module, as bytecode
- the pass pipeline
- the compile settings (environment)
- the extra link libraries
- the toolchain: the MLIR binaries

The two levels

- memory: this process, LRU if capped
- disk: lowered module + CRC32

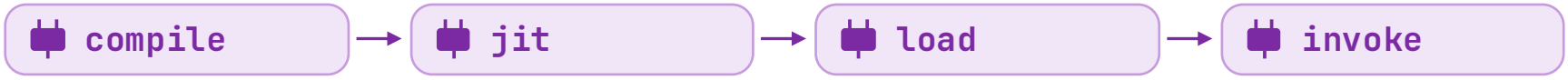
A new Python value or dtype is a new module, so a new key; the same call again is a memory hit.



Compiler plugin

Concept: The compiler plugin lowers, JITs and loads

The compiler plugin has four verbs. The pipeline behind them is the DSL's: another DSL, another pipeline.



run the pass list — build the engine — a callable for the entry — call it with the arguments

MLIRTestDSL · pipeline()

```
convert-scf-to-cf
convert-cf-to-llvm
convert-vector-to-llvm
convert-arith-to-llvm
convert-math-to-llvm
convert-func-to-llvm
reconcile-unrealized-casts
```

`execution_engine.Compiler()`

TritonDSL · its pipeline

```
triton{num_warps=4 num_stages=3}

make_ttir
ttgir
llir
ptx
cubin
```

`TritonCompiler()` (pull request)

Own compiler · no MLIR passes

TensorIRDSL · its compiler

```
pipeline() = []

nv_tensor_ir.compile
CUDA Tile IR
```

NVIDIA

`CudaTileCompiler()`

TensorIRDSL runs no MLIR pass pipeline: nv_tensor_ir.compile is NVIDIA's own compiler.

Example: Compile early, call later

m.compile traces, lowers and JITs when it is called; compiled(...) then runs without tracing again.

Python · compile early, call later



```python
@m.jit
def saxpy(a: m.Float32, x, y, n: m.Int32): ...

def async_compile(*args):
    # compiles now, in this thread
    return m.compile(saxpy, *args)

compiled = async_compile(2.0, x, y, 8)

# do bunch of other work

compiled(2.0, x, y, 8)
```

1. **m.compile**
   traces, lowers and JITs here, saves nothing
2. **Other work**
   compiled keeps the loaded function
3. **compiled(...)**
   adapts the arguments and runs: no trace

The compile is paid at the first line. Every later call only adapts the arguments and runs.



Diagnostics

(provided by BaseDSL)

Example: An error, a warning and a remark

One call each. Each one points back to your Python source. The colors are the terminal's.

```
m.report_error("tile must be > 0")
```

ERROR

```
error: tile must be > 0
  --> kernel.py:14:9
   |
14 |         m.set_tile(-1)
   |         ^^^^^^^^^^^^^^
```

```
m.report_warning("tile is too large")
```

WARNING

```
warning: tile is too large
  --> kernel.py:12:9
   |
12 |         m.set_tile(8192)
   |         ^^^^^^^^^^^^^^^^
```

```
m.report_remark("Tile size is not performant")
```

REMARK

```
remark[analysis]: Remark |
    Category:mlir.dsl | Tile size is not performant
  --> kernel.py:10:5
   |
10 |     m.set_tile(32)
   |     ^
```

Output of real runs, shortened. The report calls are the proposed API; m.set_tile is an example call.

From SSA carries to richer mutable state

Agenda 06 · Advanced mutation tracking, the story from the CUTLASS Python DSL

What if the state is more than a scalar?

(AMIR)

A hidden write changes an earlier read

Python runs the methods; MLIR carries the mutable state.

Python · ordinary state

```
class State:
    def __init__(self):
        self.c = 0              # (M) provisional
        self.bins = {"hi": 0}   # (M) provisional
        self.s = 3              # (M) read-only

    def mutate(self):           # (M) method
        self.c += self.s        # (M) → (S)
        self.bins["hi"] += 1    # (M) → (S)
```

(M) Python runs **(S) MLIR emitted**

Read c first; discover its write in mutate().

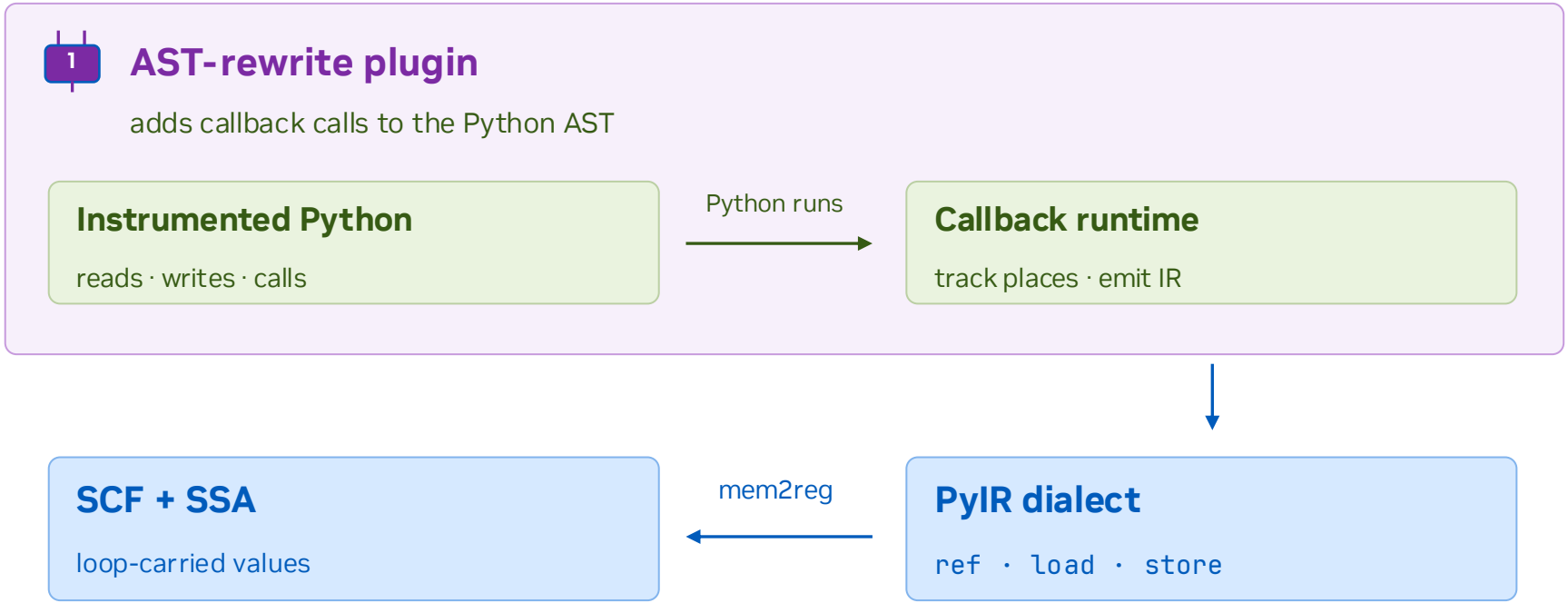
Python · runtime loop

```
@m.jit
def kernel(n: m.Int32, s):
    tot = 0                     # (M)
    for _ in m.range(n):        # (S)
        tot += s.c * s.s        # (M) → (S)
        if s.s:                 # (M)
            tot += 1            # (M) → (S)
        s.mutate()              # (M) → (S)
        m.printf(
            "c={} hi={} s={} tot={}",
            s.c, s.bins["hi"], s.s, tot,
        )

kernel(10, State())
```

The AST plugin inserts runtime callbacks

One AST-rewrite plugin: instrument Python, track mutation, emit PyIR.



The generated Python calls back into the runtime

Captured with CUTE_DSL_PRINT_AFTER_PREPROCESSOR=True.

Python · actual preprocessor output, excerpts

```
_pyir_attr_1 = __pyir_runtime__.pyir_read(
    's.c', s.c, attach_ref=False, owner=s, slot_name='c')
_pyir_attr_2 = __pyir_runtime__.pyir_read(
    's.s', s.s, attach_ref=False, owner=s, slot_name='s')
tot = __pyir_runtime__.pyir_read('tot', tot)
_pyir_old_3 = tot
tot = __pyir_runtime__._pyir_inplace_binop(
    'add', __pyir_runtime__._pyir_anchor_statement_read(tot),
    _pyir_attr_1 * _pyir_attr_2)
tot = __pyir_runtime__.pyir_assign(
    'tot', _pyir_old_3, tot, 'user-example.py', 14)
# ...
_pyir_call_boundary_(s.mutate)()
```

Track reads

c and s

Record update

read → add → write

Follow the call

instrument mutate

Trace mutable cells into MLIR

Three operations record state: ref, load and store.

MLIR · captured loop, abridged / renamed

```
%hi = pyir.ref %zero : i32
%c = pyir.ref %zero : i32
%tot = pyir.ref %zero : i32
scf.for %i = %zero to %n step %one : i32 {
  %c.read = pyir.load %c : <i32>
  %scaled = arith.muli %c.read, %three : i32
  // ... update tot, then s.mutate()
  %c.old = pyir.load %c : <i32>
  %c.new = arith.addi %c.old, %three : i32
  pyir.store %c.new, %c : <i32>
  // ... update hi; read the updated cells
  cute.print("c={} hi={} s={} tot={}\0A",
      %c.now, %hi.now, %three, %tot.now)
      : i32, i32, i32, i32
}
```

pyir.ref

Create a typed cell

pyir.load

Read its current value

pyir.store

Record an update

Which values need a cell?

Capture writes under runtime control flow.

Access	Context	Stage	Action	Python	IR pattern
Read	Before loop	Python	**Keep**	`y = s.c * 2`	`y = 0` `# folded in Python`

Write	Before loop	Python	**Keep**	`s.c += 4`
				`for i in m.range(n):`
				`  total += s.c`
					`%c4 = arith.constant 4 : i32`
					`// use %c4 in the loop`

Read	Runtime loop	Runtime	**Keep**	`for i in m.range(n):` `  total += s.c`	`%c0 = arith.constant 0 : i32` `// use %c0 in the loop`

Write	Runtime loop	Runtime	**Cell**	`for i in m.range(n):`
				`  s.c += 2`
				`  s.bins["hi"] += 1`
					`pyir.ref`
					`pyir.load → pyir.store`

Write (str)	Runtime loop	Runtime	**Reject**	`for i in m.range(n):` `  s.tag += "x"`	`PHASE_MUTATE_PYTHON`

Repair the read when the hidden write appears

s.c becomes a cell; s.s stays a literal.

Python · read before the hidden write



```
for _ in m.range(n):
    tot += s.c * s.s
    if s.s:
        tot += 1
    s.mutate()
    m.printf("c={} hi={} s={} tot={}",
             s.c, s.bins["hi"], s.s, tot)
```

Python · ordinary method



```
def mutate(self):
    self.c += self.s
    self.bins["hi"] += 1
```

MLIR · before the write, conceptual

```
%c.read = arith.constant 0 : i32
%scaled = arith.muli %c.read, %three : i32
```

Repair the earlier read



MLIR · repaired read, excerpt

```
%c = pyir.ref %zero : i32
// ... enter the generated loop
%c.read = pyir.load %c : <i32>
%scaled = arith.muli %c.read, %three : i32
```

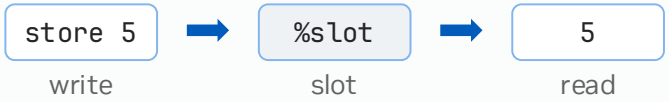
Let MLIR reconstruct SSA

Memory-slot promotion replaces cells with SSA values.

Before · memory-slot sketch

```
%slot = alloca i32
store i32 5, ptr %slot
%v = load i32, ptr %slot
use(%v)
```

Store, then load.



After · SSA sketch

```
; no slot, store or load
use(5)
```

Use the SSA value directly.



Why? Write mutable state — Optimize SSA

LLVM memory	PyIR cells
`alloca`	`pyir.ref`
`load / store`	`pyir.load / pyir.store`
`phi nodes`	`scf.for iter_args`

Mutable cells become loop carries

convert-pyir-to-scf uses MLIR memory-slot promotion.

MLIR · PyIR excerpt

```
%hi = pyir.ref %zero : i32
%c = pyir.ref %zero : i32
%tot = pyir.ref %zero : i32

scf.for %i = %zero to %n
    step %one : i32 {
  %c.read = pyir.load %c : <i32>
  // ... multiply + two adds
  // ... update c and hi
  pyir.store %c.new, %c : <i32>
  // ... load updated cells
  // ... cute.print inside loop
}
```

MLIR · captured SCF loop, renamed

```
%r:3 = scf.for %i = %zero to %n step %one
    iter_args(%hi = %zero, %c = %zero,
              %tot = %zero)
    -> (i32, i32, i32) : i32 {
  %v = arith.muli %c, %three : i32
  %t = arith.addi %tot, %v : i32
  %t1 = arith.addi %t, %one : i32
  %c1 = arith.addi %c, %three : i32
  %h1 = arith.addi %hi, %one : i32
  cute.print("c={} hi={} s={} tot={}\0A",
      %c1, %h1, %three, %t1)
      : i32, i32, i32, i32
  scf.yield %h1, %c1, %t1 : i32, i32, i32
}
```

MLIR DSL in a Glance

Ten ordinary Python features, one kernel body.

```python
class Counter:                          # plain class, plain-int fields
    def __init__(self, step):
        self.total = 0
        self.step = step
    def tick(self, x):                  # mutating method (self-mutation)
        self.total += self.step * x
    def arm(self, bonus):               # attribute born in a method
        self.boost = bonus

class Weighted(Counter):                # subclass
    def tick(self, x):
        super().tick(x)                 # inheritance with super()
        self.total += 1

@cute.jit
def kernel(n: cutlass.Int32):
    a = Counter(1)                      # attribute state through the loop
    a.arm(11)                           # born on the fly, before the loop
    w = Weighted(2)                     # subclass instance
    objs = [Counter(3), Counter(5)]     # trace-time list of objects
    stats = {"acc": 0, "hits": 0}       # keys exist before the loop
    calls = 0
    def note():                         # closure with nonlocal
        nonlocal calls
        calls += 1
    peek = lambda: a.total + calls      # lambda, read after the loop
    for i in range(n):                  # trip count is the runtime arg
        a.tick(i)                       # mutating method
        a.boost += i                    # the on-the-fly field, per trip
        w.tick(i)                       # subclass method into super()
        for o in objs:                  # trace-time iteration
            o.total += o.step * i       # per-iteration attr mutation
        stats["acc"] += i * i           # dict value mutation
        if i % 2 == 1:                  # runtime if inside the loop
            stats["hits"] += 1          # conditional dict update
        note()                          # closure called in the loop
```

TEN FEATURES, ONE BODY

1. Attribute state carried through the loop
2. Mutating methods, self-mutation included
3. Inheritance, with super()
4. A field born outside __init__
5. A trace-time list of objects
6. Dict value updates
7. A runtime if in the loop
8. A closure with nonlocal
9. A lambda read after the loop
10. A runtime trip count

Plain Python and MLIR DSL

Same output

The frontend for RTS, CTM and CuTe DSL kernels.

MLIRTest DSL

Agenda 07 · example 1 of 4

The shipped composition: all five plugin types, a @kernel call and three programs.

Example: `class MLIRTestDSL(BaseDSL)`

A name and a Plugins record with five plugin types make a working sub-DSL.

MLIRTestDSL · shipped plugin record

```
# Step 1. Create My end-2-end DSL
class MLIRTestDSL(BaseDSL):
    # Step 2. Setup my plugins
    plugins = Plugins(
        # Step 3. Insert Type Plugin
        type_ops=UpstreamDialectTypeOps(
            scalars=arith, vectors=vector, memory=llvm),
        # Step 4. Insert AST-preprocessor
        ast_preprocessor=scf.ASTPreprocessor(),
        # Step 5. Insert Compiler
        compiler=execution_engine.Compiler(),
        # Step 6. Insert Decorators
        decorators=[func.Jit(), gpu_plugin.Kernels(
            chip_option="cubin-chip")],
        # Step 7. Insert Adapters
        adapters=[dlpack.DlpackPlugin(), tvm_ffi.TvmFfiPlugin()],
    )
```

The plugins it names

- Step 3 — **Types plugin** — `UpstreamDialectTypeOps`
- Step 4 — **AST plugin** — `scf.ASTPreprocessor`
- Step 5 — **Compiler plugin** — `execution_engine.Compiler`
- Step 6 — **Decorator plugins** — `func.Jit` `gpu_plugin.Kernels`
- Step 7 — **Adapter plugins** — `DlpackPlugin` `TvmFfiPlugin`

Example: A Python callback in a GPU kernel

The same scalar function, passed as a Python value, is called while the kernel is traced

Python · user code



```python
@m.kernel
def map_kernel(op, a: m.Float32, x, y, n: m.Int32):
    for i in m.range(n):
        y[i] = op(a, x[i], y[i])
@m.jit
def saxpy_gpu(a: m.Float32, x, y, n: m.Int32):
    map_kernel(axpy, a, x, y, n).launch(
        grid=[1], block=[1])
```

Generated MLIR

```
gpu.func @map(...) kernel {
  // op is not an IR operand
  scf.for %i = %c0 to %n {
    %ax = arith.mulf %a, %xi
    %v = arith.addf %ax, %yi
    // loads and store omitted
  }
}
func.func @saxpy_gpu(...) {
  gpu.launch_func @kernels::@map
}
```

op is a Python value, not an IR operand; GPU IR verified (trace only)

EmitC DSL

Agenda 07 · example 2 of 4

The same Python and dtypes with another dialect: two TypeOps hooks answer with EmitC.

Example: `class EmitCTestDSL(BaseDSL)`

The same record with one plugin of its own: type_ops answers add and mul with EmitC.

EmitCTestDSL · the plugin record

```
# Step 1. Create My end-2-end DSL
class EmitCTestDSL(BaseDSL):
    # Step 2. Setup my plugins
    plugins = Plugins(
        # Step 3. Insert Type Plugin
        type_ops=EmitCScalarOps(
            scalars=arith, vectors=vector, memory=llvm),
        # Step 4. Insert AST-preprocessor
        ast_preprocessor=scf.ASTPreprocessor(),
        # Step 5. No compiler: the traced IR is the product
        # Step 6. Insert Decorators
        decorators=[func.Jit()],
        # Step 7. No adapters
    )
```

The plugins it names

- **Step 3** — Types plugin: `EmitCScalarOps`
- **Step 4** — AST plugin: `scf.ASTPreprocessor`
- **Step 5** — Compiler plugin: none: the traced IR is the product
- **Step 6** — Decorator plugin: `func.Jit`
- **Step 7** — Adapter plugin: none

Example: class EmitCScalarOps(UpstreamDialectTypeOps)

Same Python and dtypes; mlir.emitc_dsl answers two hooks with EmitC.

Python · same scalar body



```
@e.jit
def axpy(a: e.Float32, x: e.Float32,
         y: e.Float32):
    return a * x + y
```

Plugin · EmitCScalarOps



```
class EmitCScalarOps(UpstreamDialectTypeOps):
    def add(self, lhs, rhs, *, signed=None, loc=None, ip=None):
        return emitc.add(lhs.type, lhs, rhs, loc=loc, ip=ip)
    def mul(self, lhs, rhs, *, signed=None, loc=None, ip=None):
        return emitc.mul(lhs.type, lhs, rhs, loc=loc, ip=ip)
```

Generated MLIR

```
func.func @axpy(%arg0: f32, %arg1: f32,
                %arg2: f32) -> f32
    attributes {llvm.emit_c_interface} {
  %0 = emitc.mul %arg0, %arg1
       : (f32, f32) -> f32
  %1 = emitc.add %0, %arg2
       : (f32, f32) -> f32
  return %1 : f32
}
```

Output · mlir-translate --mlir-to-cpp

```
float axpy(float v1, float v2, float v3) {
  float v4 = v1 * v2;
  float v5 = v4 + v3;
  return v5;
}
```

NVIDIA’s TensorIR DSL

Agenda 07 · example 3 of 4

A DSL whose only value is the whole tensor, and only the AST plugin role stays empty.

Example: `class TensorIRDSL(BaseDSL)`

One role stays empty. The types plugin makes the tensor operators, the compiler is NVIDIA's own.

TensorIRDSL · the record

```
# Step 1. Create My end-2-end DSL
class TensorIRDSL(BaseDSL):
    # Step 2. Setup my plugins
    plugins = Plugins(
        # Step 3. Insert Type Plugin
        type_ops=TensorOps(),
        # Step 4. No AST-preprocessor: no control flow
        # Step 5. Insert Compiler
        compiler=CudaTileCompiler(tile_sizes=(64, 64)),
        # Step 6. Insert Decorators
        decorators=[GraphKernels()],
        # Step 7. Insert Adapters
        adapters=[DlpackTensors()],
    )
```

CudaTileCompiler calls NVIDIA's Tensor IR compiler: nv_tensor_ir.compile lowers the graph to CUDA Tile IR.

The plugins it names

- **Step 3** — Types plugin: `TensorOps`
- **Step 4** — AST plugin: none: a graph has no control flow
- **Step 5** — Compiler plugin: `CudaTileCompiler` NVIDIA
- **Step 6** — Decorator plugin: `GraphKernels`
- **Step 7** — Adapter plugin: `DlpackTensors`

Example: A fused GEMM epilogue on Tensor IR

The public Tensor IR DSL's README kernel on the sub-DSL: no annotations, a Python if, one graph

Python · user code



```python
import tensor_ir_dsl as tir

@tir.kernel
def fused_gemm_epilogue(a, b, bias, res, GELU):
    x = a @ b + bias
    if GELU :    # decided while tracing
        x = tir.gelu_approx_tanh(x)
    return x + res  # the graph's results

a = torch.randn(8, 32, device="cuda")
b = torch.randn(32, 16, device="cuda")
bias = torch.randn(8, 16, device="cuda")
res = torch.randn(8, 16, device="cuda")
out = fused_gemm_epilogue(a, b, bias, res, TRUE)
```

Generated MLIR

```
nv_tensor_ir.graph @fused_gemm_epilogue(
    %a: tensor<8x32xf32>, %b: tensor<32x16xf32>,
    %bias: tensor<8x16xf32>,
    %res: tensor<8x16xf32>)
    -> (tensor<8x16xf32>) {
  %matmul = matmul(%a, %b) : (tensor<8x32xf32>,
      tensor<32x16xf32>) -> tensor<8x16xf32>
  %add = add %matmul, %bias : tensor<8x16xf32>
  %gelu = gelu_approx_tanh_fwd %add
      : tensor<8x16xf32>
  %add_3 = add %gelu, %res : tensor<8x16xf32>
  results %add_3 : tensor<8x16xf32>
}
```

Traced from torch tensors; tensor_ir-opt checks the graph and lowers it to CUDA Tile IR. Trace only: no launch, out is None.

OpenAI’s Triton DSL

Agenda 07 · example 4 of 4 · Amir Mohammad Tavakkoli

Triton's frontend reads your Python;
here mlir.dsl runs it, and the stock
Triton compiler does the rest.

Example: `class TritonDSL(BaseDSL)`

Every role is filled. The compiler plugin hands the TTIR to the stock Triton compiler.

TritonDSL · the record

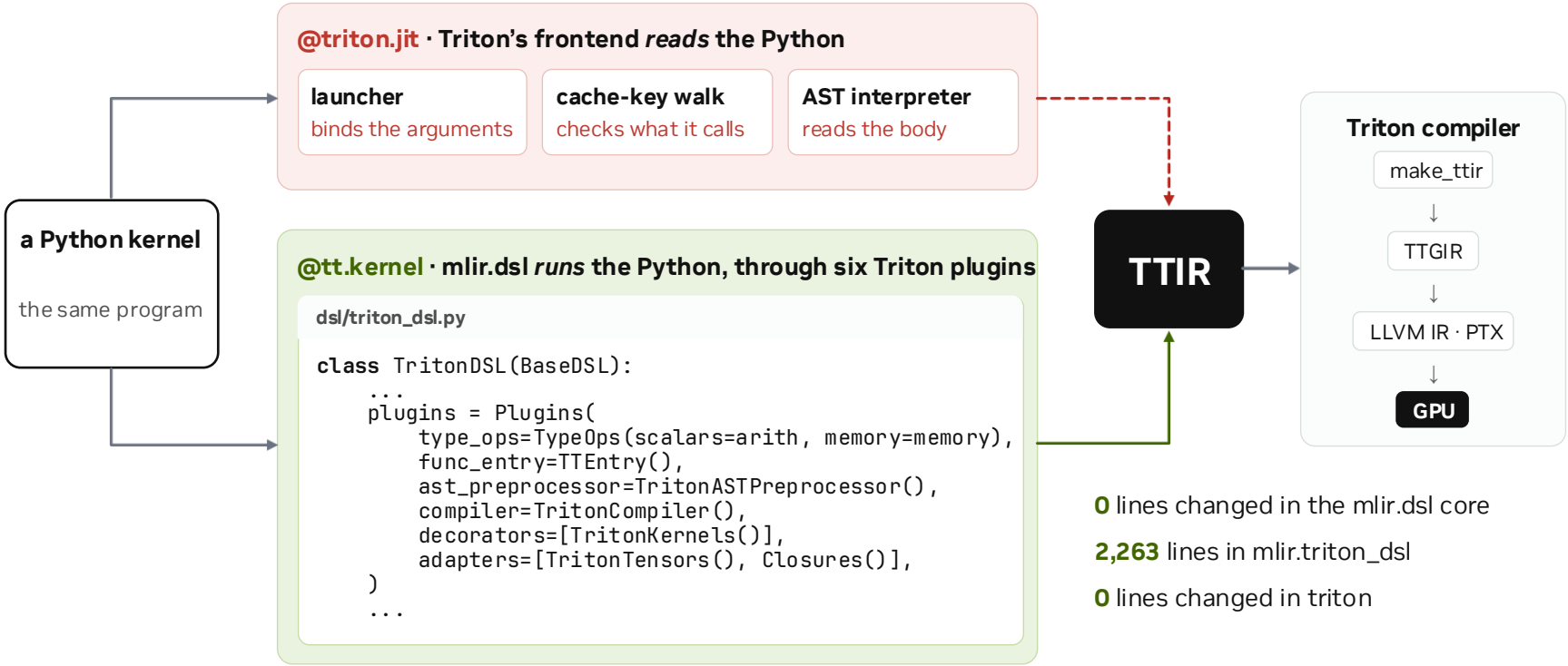
```
# Step 1. Create My end-2-end DSL
class TritonDSL(BaseDSL):
    # Step 2. Setup my plugins
    plugins = Plugins(
        # Step 3. Insert Type Plugin
        type_ops=TypeOps(scalars=arith, memory=memory),
        # Step 4. Insert AST-preprocessor
        ast_preprocessor=TritonASTPreprocessor(),
        # Step 5. Insert Compiler
        compiler=TritonCompiler(),
        # Step 6. Insert Decorators
        decorators=[TritonKernels()],
        # Step 7. Insert Adapters
        adapters=[TritonTensors(), Closures()],
    )
```

The plugins it names

- **Step 3** — Types plugin: `TypeOps`
- **Step 4** — AST plugin: `TritonASTPreprocessor`
- **Step 5** — Compiler plugin: `TritonCompiler`
- **Step 6** — Decorator plugin: `TritonKernels`
- **Step 7** — Adapter plugins: `TritonTensors` `Closures`

From Amir's pull request, written for an older core: TypeOps is now UpstreamDialectTypeOps, and TTEntry's job is the decorator's.

OpenAI Triton DSL



Your Triton kernel, almost unchanged

tl. becomes tt.; only the runtime scalar N needs a type (blue). Tensors become pointers, Python values are compile-time. One Triton compiler.

@triton.jit **runs**

```
@triton.jit
def softmax(src, dst, N, BLOCK: tl.constexpr):
    row = tl.program_id(0)
    cols = tl.arange(0, BLOCK)
    x = tl.load(src + row * N + cols, cols < N,
                float("-inf"))
    e = tl.exp(x - x.max())
    tl.store(dst + row * N + cols, e / e.sum(),
             cols < N)

softmax[(rows,)](x, y, N, 1024)
```

@tt.kernel **runs**

```
@tt.kernel
def softmax(src, dst, N: tt.Int32, BLOCK):
    row = tt.program_id(0)
    cols = tt.arange(0, BLOCK)
    x = tt.load(src + row * N + cols, cols < N,
                float("-inf"))
    e = tt.exp(x - x.max())
    tt.store(dst + row * N + cols, e / e.sum(),
             cols < N)

softmax[(rows,)](x, y, N, 1024)
```

Nothing to give up. What you gain is the rest of Python inside the kernel.

Five kinds of Python that @triton.jit rejects

1 **Functions as values**
lambdas, closures, partials and bound methods passed to a kernel

2 **Objects and classes**
records, methods, dispatch and user types as kernel arguments

3 **Flexible signatures**
*args, **kwargs, and kernels that are methods of a class

4 **Python data structures and control flow**
lists, dicts, comprehensions, match, try and recursion in the kernel

5 **Kernels generated from Python data**
an expression graph, an autodiff tape, a storage format

Functions as values

1 / 5

Pass functions into kernels

Closures as kernel arguments



```
def leaky(slope):
    return lambda v, offs: tt.where(v > 0, v, v * slope)

def scaled(alpha):
    a = tt.Float32(alpha)
    return lambda v, offs: v * a

def add_bias(b):
    return lambda v, offs: v + tt.load(b + offs)

@tt.kernel
def apply(src, dst, fn):
    offs = tt.program_id(0) * 1024 + tt.arange(0, 1024)
    tt.store(dst + offs, fn(tt.load(src + offs), offs))

apply[(4,)](x, y, leaky(0.1))
apply[(4,)](x, y, scaled(0.5))
apply[(4,)](x, y, add_bias(bias))
```

@triton.jit · Triton runs the lambda as plain Python, so its tile operation fails.

FlexAttention mods in Python



```
def sliding_window(w):
    return lambda qi, ki: (qi >= ki) & (qi - ki < w)

def alibi(slopes):
    return lambda s, h, qi, ki: (s - tt.load(slopes + h)
                                 * (qi - ki))

@tt.kernel
def attention(q, k, v, o, score_mod, mask_mod):
    ...
    s = score_mod(s, h, i[:, None], i[None, :])
    s = tt.where(mask_mod(i[:, None], i[None, :]), s, -1e30)
    ...
    tt.store(o + at, tt.dot(p.to(tt.Float16), tt.load(v + at)))

attention[(H,)](q, k, v, o, alibi(slopes), sliding_window(16))
```

@triton.jit · Triton runs score_mod as plain Python; it fails on tiles.

Halide-style schedules

```
def blur(load, x):
    return (load(x - 1) + load(x) + load(x + 1)) / 3.0

...

@tt.kernel
def fused(src, dst, outer, inner):
    x = tt.arange(0, 1024)
    def inside(x):
        ok = (x >= 0) & (x < 1024)
        return tt.where(ok, inner(reader(src), x), 0.0)
    tt.store(dst + x, outer(inside, x))

class Fused:
    def run(self, src, dst):
        fused[(1,)](src, dst, blur, blur)

...

for schedule in (Fused(), TwoPass()):
    schedule.run(x, y)
```



@triton.jit · Triton refuses reader, a plain Python function the kernels call.

Objects and classes

2 / 5

Your own types as kernel arguments

A user-defined tensor class



```
@dataclass(frozen=True)
class Tensor:
    ...
    def __getitem__(self, index):
        return tt.load(self.at(*index))
    def __setitem__(self, index, value):
        tt.store(self.at(*index), value)
    @property
    def T(self):
        return Tensor(self.data, self.strides[::-1])

...

@tt.kernel
def add_transposed(x, y, out):
    ...
    out[r, c] = x[r, c] + y.T[r, c]

add_transposed[(4, 4)](of(x), of(y), of(out))
```

@triton.jit · Triton's launcher can't specialise the user's Tensor class.

Dispatch on operand class



```
...
@functools.singledispatch
def load_tile(op, offs):
    raise TypeError(type(op).__name__)

...
@load_tile.register
def _(op: Int8Scaled, offs):
    return tt.load(op.values + offs).to(tt.Float32) * op.scale

...
    tt.store(dst + offs,
             load_tile(a, offs) + load_tile(b, offs))
for b in (Dense(y), Int8Scaled(q, tt.Float32(0.05))):
    add[(4,)](Dense(x), b, out)
```

@triton.jit · Triton's launcher can't specialise the operand dataclass.

Pallas-style index maps



```
@tt.jit
def block(p, coords, N):
    ...
@tt.kernel
def matmul(a, b, c, a_map, b_map, N):
    ...
    for k in range(N // 32):
        acc = tt.dot(block(a, a_map(i, j, k), N),
                     block(b, b_map(i, j, k), N), acc)
    ...

...
experts = lambda i, j, k: (tt.load(route + i) * 4 + k, j)
matmul[(4, 4)](a, w, c, lambda i, j, k: (i, k), experts, N)
```

@triton.jit · Triton runs the index map as Python; its tuple can't enter a jit helper.

Flexible signatures

3 / 5

Variadic and class-based kernels

Variadic kernel inputs



```
@tt.kernel
def map_n(fn, dst, *inputs):
    offs = tt.program_id(0) * 1024 + tt.arange(0, 1024)
    tt.store(dst + offs,
             fn(*[tt.load(p + offs) for p in inputs]))

@tt.kernel
def evaluate(formula, dst, **named):
    offs = tt.program_id(0) * 1024 + tt.arange(0, 1024)
    values = {k: tt.load(p + offs) for k, p in named.items()}
    tt.store(dst + offs, formula(**values))

map_n[(4,)](lambda a, b: a * b, y, x0, x1)
map_n[(4,)](lambda a, b, c: a * b + c, y, x0, x1, x2)
evaluate[(4,)](lambda x, w: x * w, y, x=x0, w=x1)
```

@triton.jit · Triton's launcher has a fixed signature: *inputs is one parameter.

Kernels as classes



```
@dataclass(frozen=True)
class Scale:
    ...
    @tt.kernel
    def body(self, src, dst):
        offs = tt.program_id(0) * 1024 + tt.arange(0, 1024)
        tt.store(dst + offs,
                 self.transform(tt.load(src + offs)))

@dataclass(frozen=True)
class Clipped(Scale):
    limit: float = 1.0
    def transform(self, v):
        v = super().transform(v)
        ...

Scale(k=3.0).body[(4,)](x, y)
Clipped(k=3.0, limit=0.5).body[(4,)](x, y)
```

@triton.jit · Triton doesn't bind the kernel method to its instance: dst is missing.

Real Python inside the kernel

Lists of tiles



```
@tt.kernel
def softmax(src, dst, N: tt.Int32, CHUNK, CHUNKS):
    row = tt.program_id(0)
    cols = [c * CHUNK + tt.arange(0, CHUNK)
            for c in range(CHUNKS)]
    chunks = [tt.load(src + row * N + c, c < N, float("-inf"))
              for c in cols]
    peak = max([ch.max() for ch in chunks])
    exps = [tt.exp(ch - peak) for ch in chunks]
    total = sum(e.sum() for e in exps)
    for c, e in zip(cols, exps):
        tt.store(dst + row * N + c, e / total, c < N)

softmax[(256,)](x, y, 781, 256, 4)
```

@triton.jit · Triton accepts comprehensions only over tuple literals.

An interpreter in the kernel



```
...
@tt.kernel
def run(src, dst, program):
    ...
    for op in program:
        match op:
            case Scale(k):
                v = v * k
            case Shift(c):
                v = v + c
            case Relu():
                v = tt.maximum(v, 0.0)
    tt.store(dst + offs, v)

run[(4,)](x, y, (Scale(0.5), Shift(1.0), Relu()))
```

@triton.jit · Triton loops only over range; a loop over the ops tuple crashes it.

Fused time steps



```
...
@tt.kernel
def steps(src, dst, rule, T):
    x = tt.program_id(0) * 256 + tt.arange(0, 256)
    memo = {}
    def u(t, dx):
        if (t, dx) not in memo:
            if t == 0:
                memo[t, dx] = tt.load(src + (x + dx + N) % N)
            else:
                left, mid = u(t - 1, dx - 1), u(t - 1, dx)
                memo[t, dx] = rule(left, mid, u(t - 1, dx + 1))
        return memo[t, dx]
    tt.store(dst + x, u(T, 0))

steps[(16,)](a, b, heat, 3)
```

@triton.jit · Triton's AST interpreter has no dicts, so the memo is refused.

Kernels generated from Python data

5 / 5

Kernels built from Python data

Graph in, fused kernel out



```
@dataclass(frozen=True, eq=False)
class Node:
    fn: object
    args: tuple
    def __mul__(a, b):
        return Node(lambda x, y: x * y, (a, b))
    ...
@tt.kernel
def fused(expr, out):
    row, c = tt.program_id(0), tt.arange(0, 256)
    def emit(n):
        ...
        return n.fn(*[emit(a) for a in n.args])
    tt.store(out + row * 256 + c, emit(expr))
X = Node(None, (x,))
rms = X * ((X * X).sum() * (1 / 256) + 1e-6).rsqrt()
fused[(64,)](rms, y)
```

@triton.jit · Triton's launcher can't specialise the graph object.

Backward from the forward



```
class Var:
    ...
    def exp(a):
        y = tt.exp(a.v)
        return Var(y, a.tape, ((a, lambda g: g * y),))
def vjp(f, x, dy):
    ...
    for node, back in reversed(tape):
        for parent, rule in back:
            g = rule(node.g)
            ...
@tt.kernel
def backward(f, x, dy, dx):
    offs = tt.program_id(0) * 1024 + tt.arange(0, 1024)
    g = vjp(f, tt.load(x + offs), tt.load(dy + offs))
    tt.store(dx + offs, g)
backward[(4,)](lambda x: x * x.exp(), x, dy, dx)
```

@triton.jit · Triton refuses vjp, a plain Python function the kernel calls.

Conclusion: A DSL is a Plugins record on one shared core

Agenda 08 · A new DSL names the hooks that differ; the core does the rest.

BaseDSL

Dialect-independent core

Tracing
Type inference
Diagnostics
JIT cache

+

Plugins record · what a DSL names

`DecoratorPlugin`	your decorator: call, launch and entry
`ASTPreprocessorPlugin`	staged for, if and while
`AdapterPlugin`	host objects as arguments
`TypeOpsPlugin`	your dialect's types and operations
`CompilerPlugin`	lower, JIT, load and invoke

Four DSLs on the same core, and the dialect each one generates

MLIRTest DSL
`arith · scf · func`

EmitC DSL
`emitc`

NVIDIA's TensorIR DSL
`nv_tensor_ir`

OpenAI's Triton DSL
`tt`