CUTLASS Python DSL Infrastructure

Guray Ozen | LLVM Developers Conference | 30th October 2025

CuTeDSL

How will you get started?

```
$>pip install nvidia-cutlass-dsl
```

```python
import cutlass.cute as cute

@cute.kernel
def kernel():
    tidx, _, _ = cute.arch.thread_idx()
    if tidx == 0:
        cute.printf("Hello world\n")

@cute.jit
def host():
    kernel().launch(grid=(1, 1, 1), block=(32, 1, 1))

host()
```

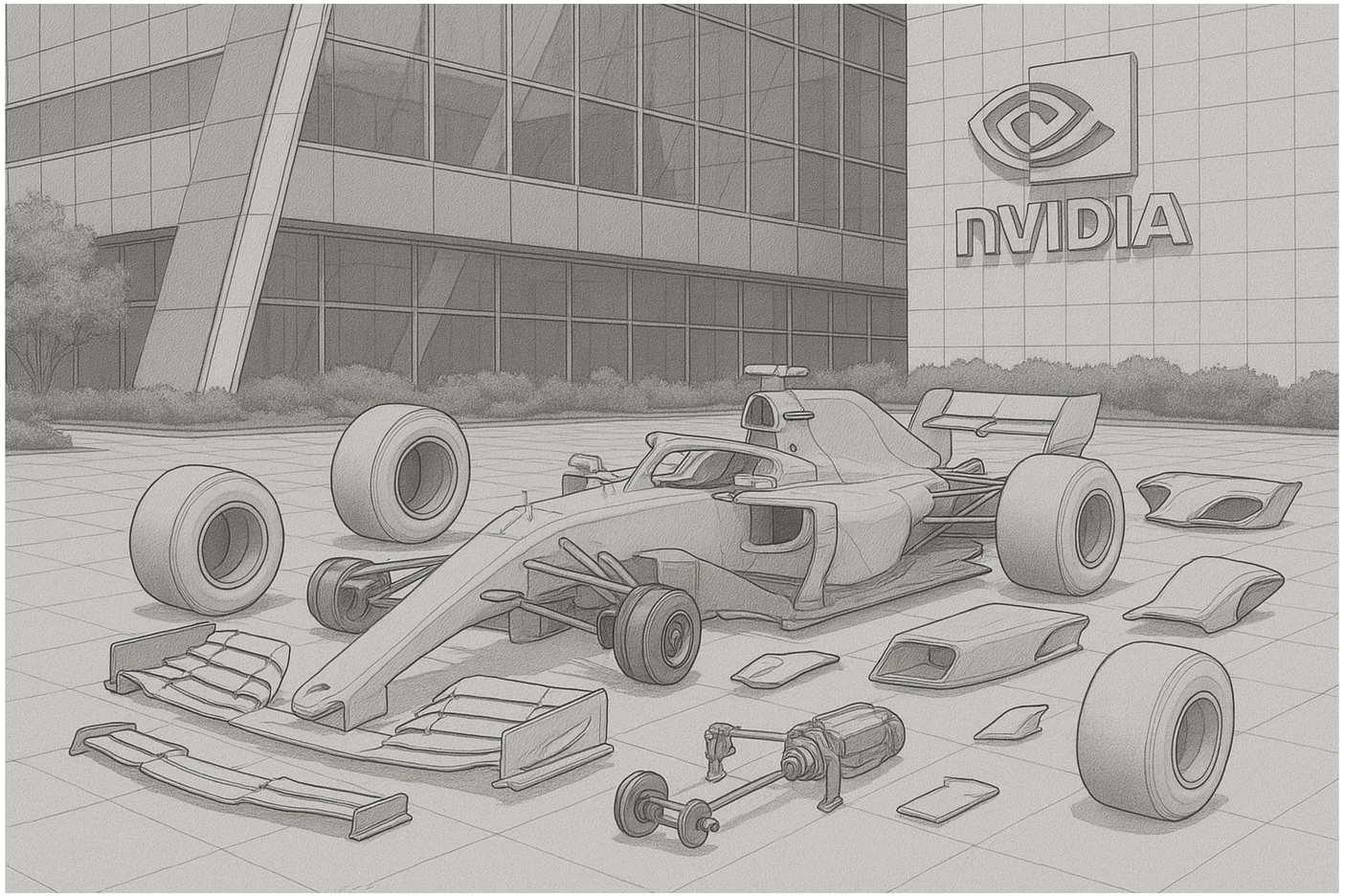
```
$>python hi.py
```

CUTLASS Python DSL

How Fast is the DSL?

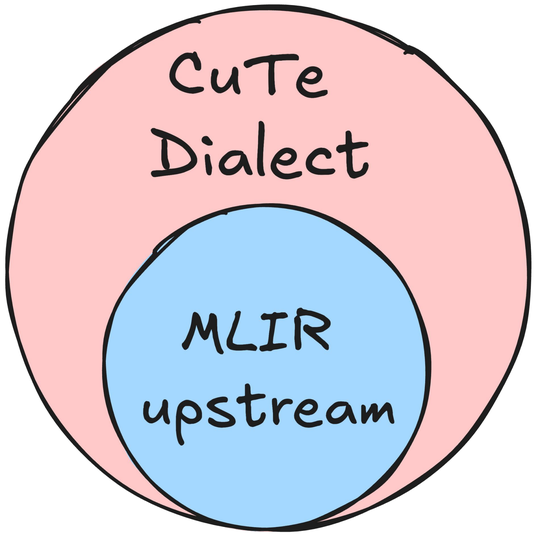
Design of CUTLASS Python DSL

We discuss how we build the DSL



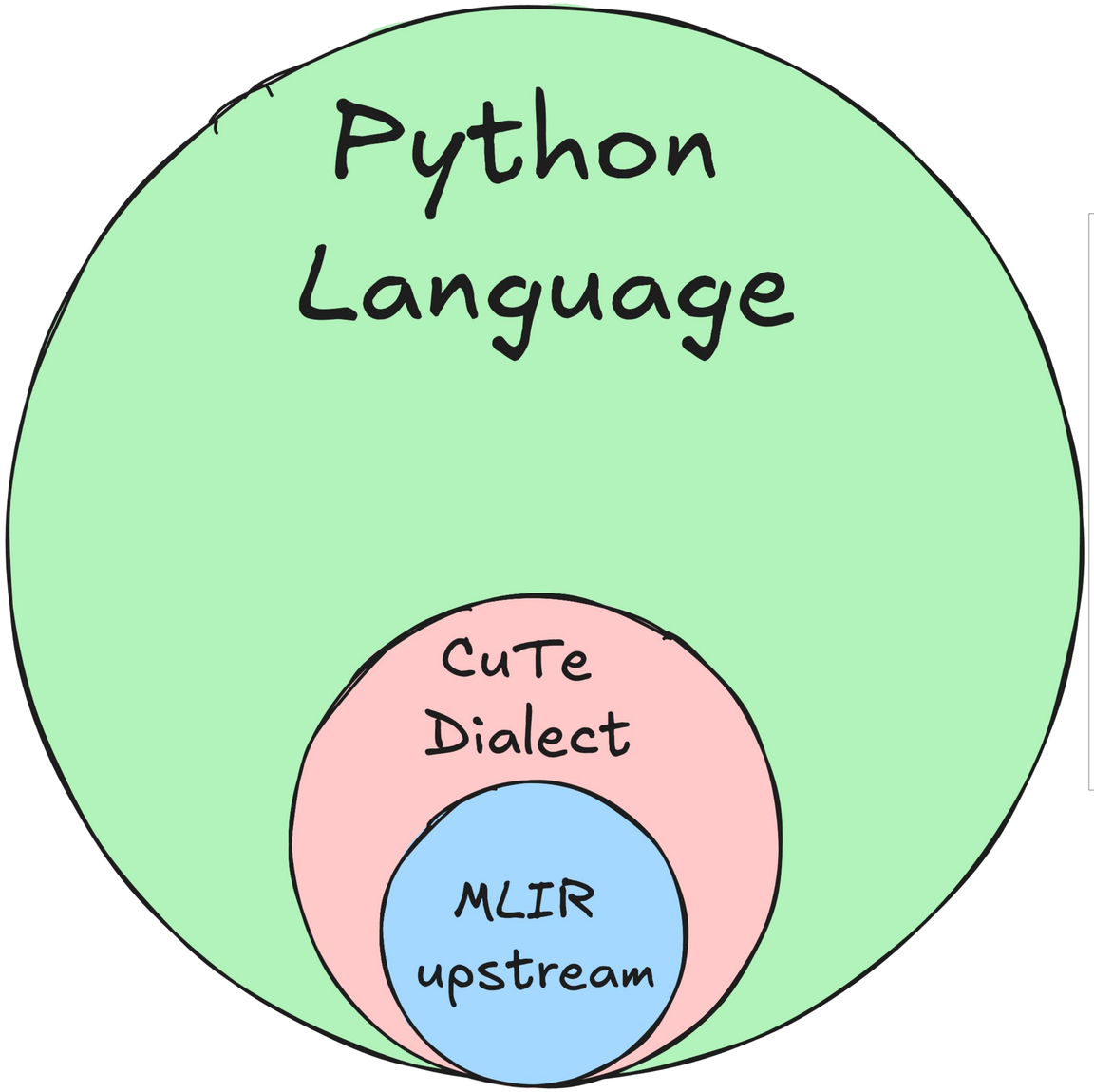
How Can I Implement a Python DSL?

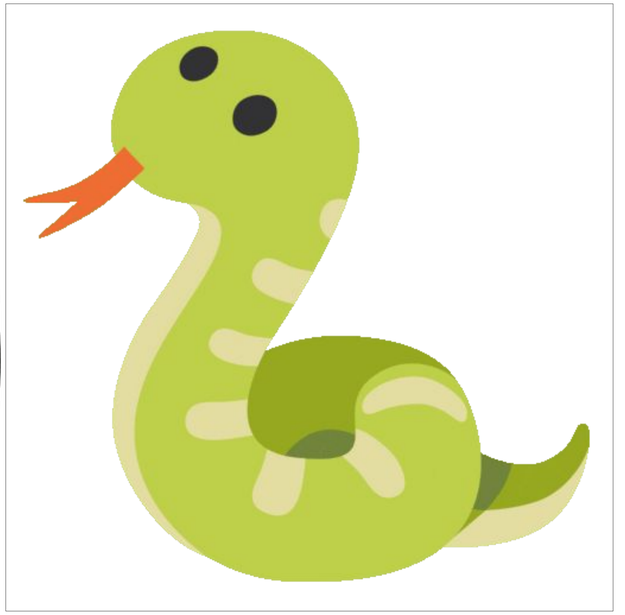
How Can I Implement a Python DSL?



How Can I Implement a Python DSL?

Python is huge… but your IR isn’t.





How to capture **Python** to MLIR?

~~How to capture **Python** to MLIR?~~

Which Python Unlocks Rapid GPU Kernel Authoring?

without sacrificing the performance

Python DSL

AST vs Bytecode vs Tracing — What to Choose?

Which Python Layer Should We Capture?

1. **Python AST**
   - Captures source structure before execution
   - Access to full program structure (loops, control flow, variables)
   - Harder to deal with dynamic runtime values

2. **Python Bytecode**
   - Closer to Python interpreter semantics
   - Stable, well-defined format
   - Loses some high-level structure (e.g., syntax, types)

3. **Tracing**
   - Captures what actually runs at runtime
   - Great for dynamic shapes & specialization
   - Limited control flow visibility and harder debugging

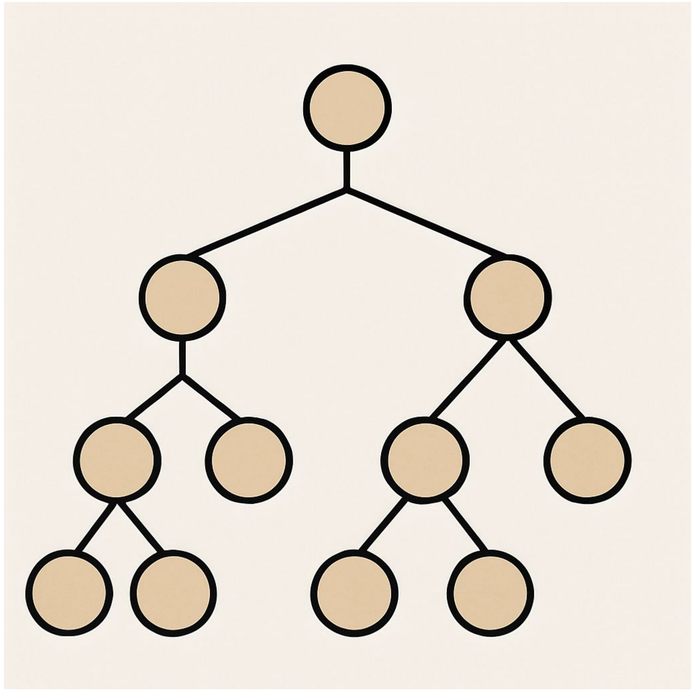
[Option 1] : AST-based Python DSL

import ast

Python DSL

```
@preciousDSL.jit
def kernel(A, B, C):

  for i in range(k):
    C = gemm(A[...], B[...], C)

  if relu:
    C = do_relu(C)
```

Python AST



Precious
MLIR Dialect

```
precious.func @kernel(%A, %B, %C, %k):
{
    precious.for %i = %c0  to %k, step %c1:
      %fA = precious.load(%A, ...)
      %fC = precious.load(%B, ...)
       %C = precious.gemm(%fA, %fB, %C)

    precious.if %relu:
       %C = precious.call(@do_relu, %C)
}
```

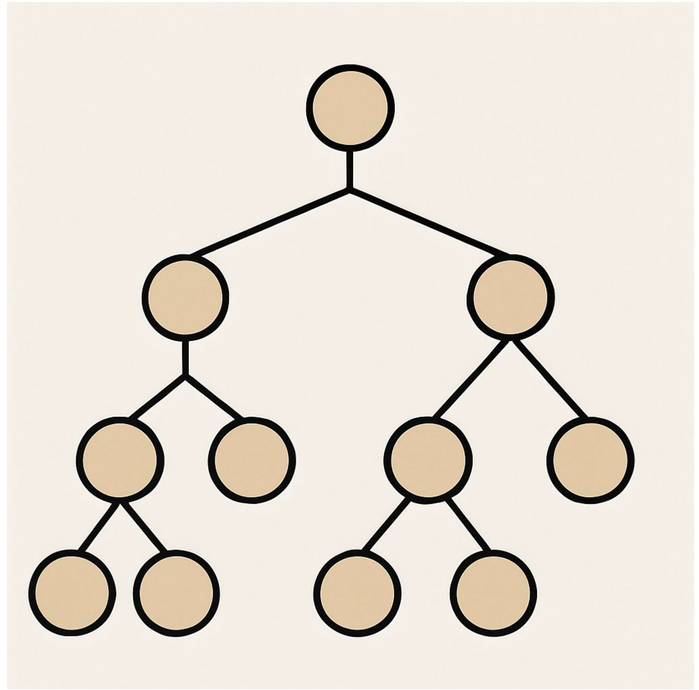
[Option 1] : AST-based Python DSL

import ast

Python DSL

```
@preciousDSL.jit
def kernel(A, B, C):

  for i in range(k):
    C = gemm(A[...], B[...], C)

  if relu:
    C = do_relu(C)
```

Python AST



Precious MLIR Dialect

```
precious.func @kernel(%A, %B, %C, %k):
{
    precious.for %i = %c0  to %k, step %c1:
       %fA = precious.load(%A, ...)
       %fC = precious.load(%B, ...)
        %C = precious.gemm(%fA, %fB, %C)

    precious.if %relu:
        %C = precious.call(@do_relu, %C)
}
```

Example: AST-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
   for i in range(C.shape[1]):
       matrix = gemm(A, B, C, i)
   Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Example: AST-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
   for i in range(C.shape[1]):
       matrix = gemm(A, B, C, i)
   Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Example: AST-Based DSL

Fused Activation Functions

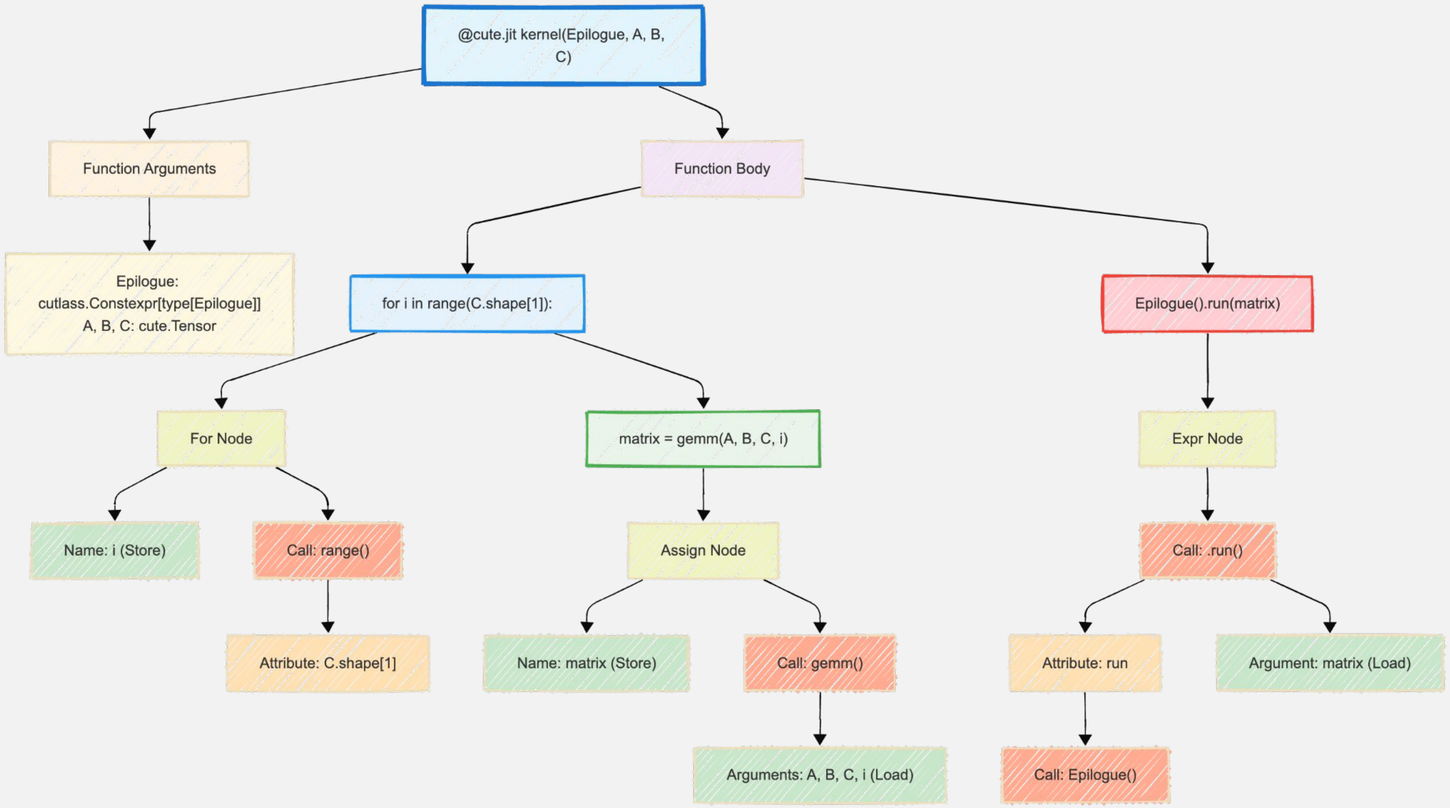
```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Example: AST-Based DSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Python AST

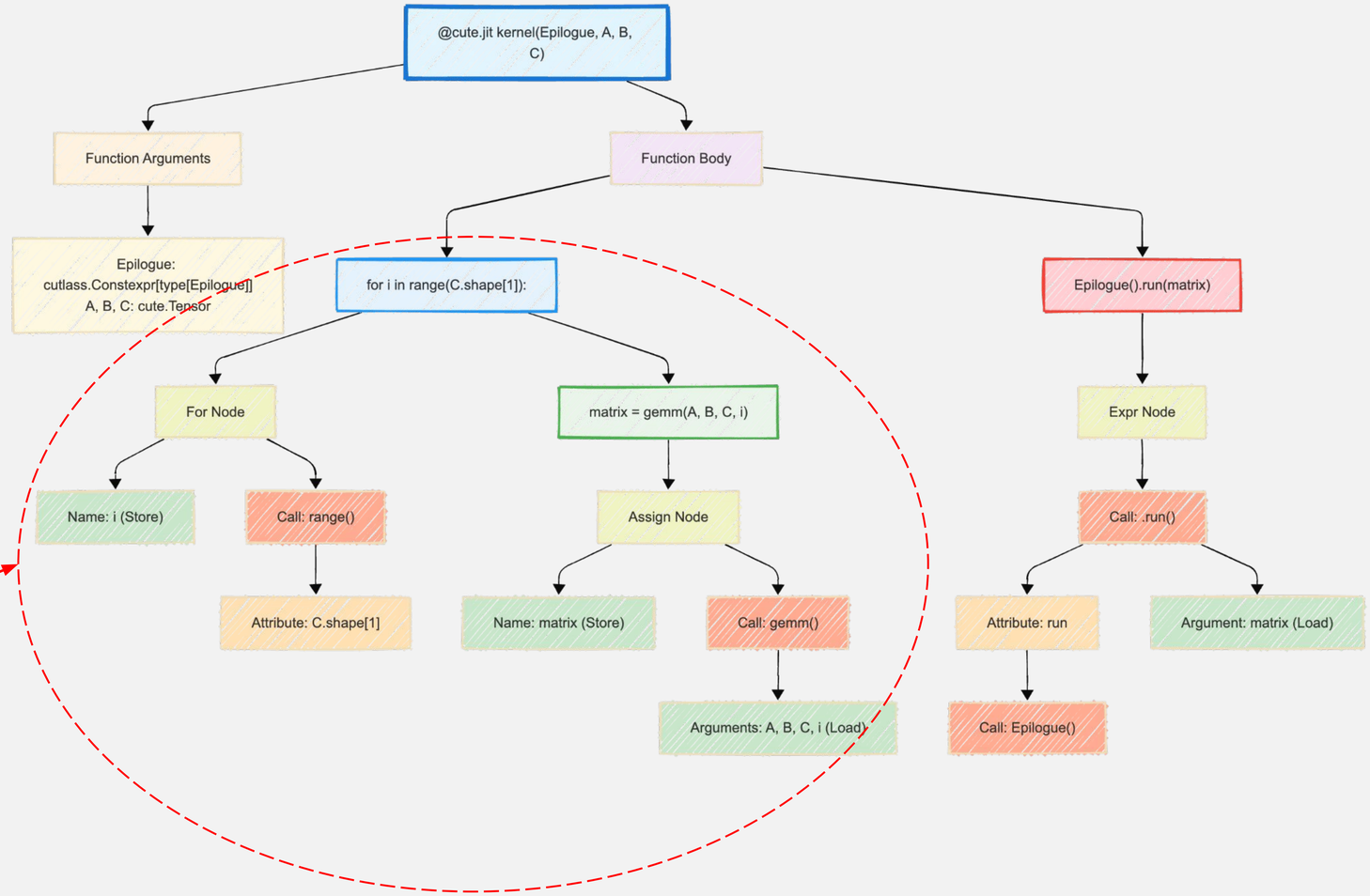


Example

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Python AST

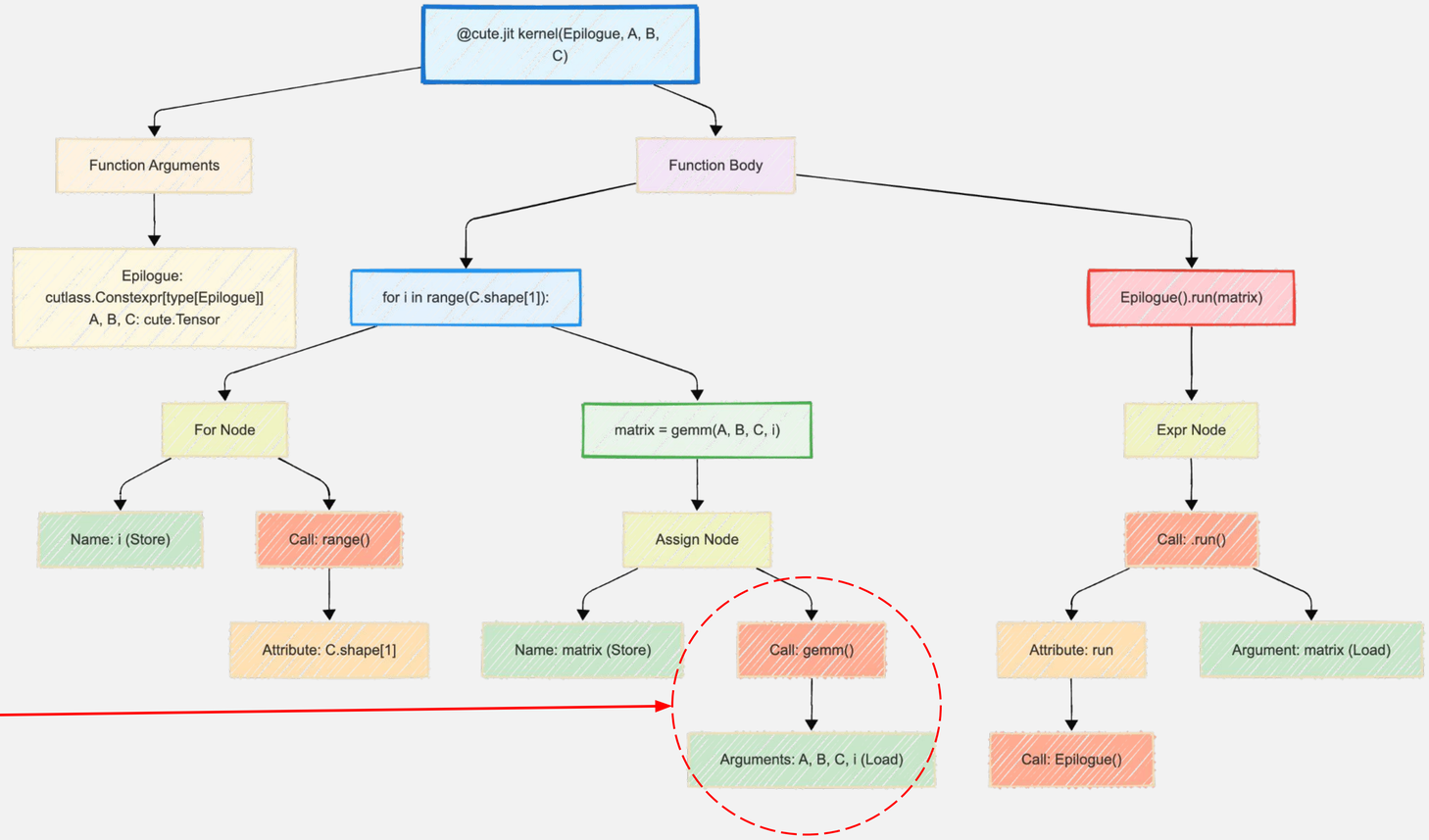


Example

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Python AST

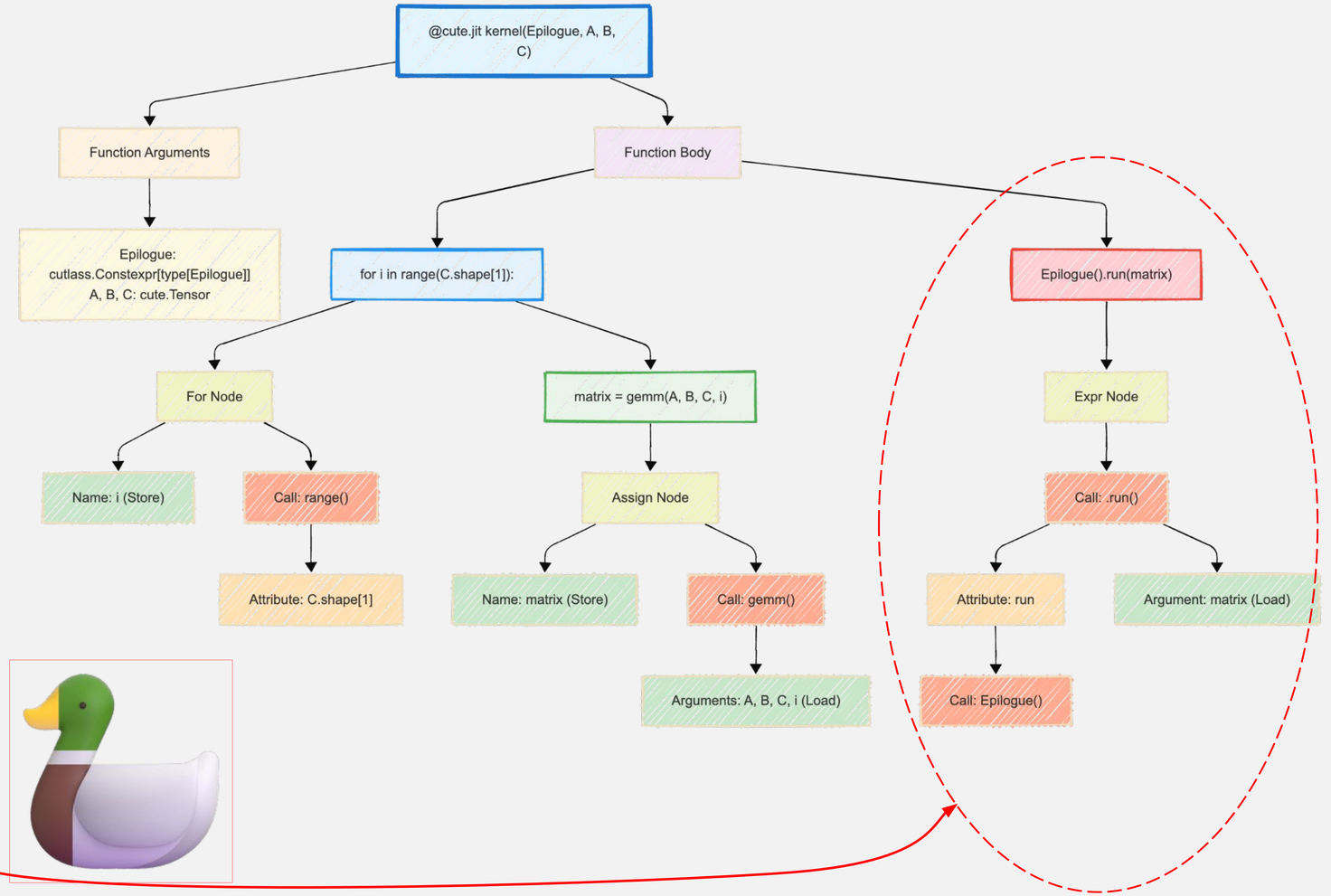


Example

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Python AST



Does AST-Based DSL Unlock Rapid GPU Kernel Authoring?

Users must “think in DSL,” not Python

- **Advantages**
  - **Loops and variables are explicit** — easy to analyze and transform
  - **Clear structure** —predictable lowering to IR
  - **Nice error messages** — tied to source locations and syntax

- **Disadvantages**
  - **Not real Python** — diverges from standard runtime semantics
  - **No Metaprogramming** — requires heavy AST infrastructure
  - **Limited flexibility** — harder to mix with other Python features
  - **DSL subset evolves slowly** — adding features is expensive
  - Each new Python construct needs a lowering rule

[Option 3] : Tracing-based Python DSL

Trace Python object and record IR

- **Core Idea**
  - Run the Python function once with proxy objects (tracers)
  - Record operations into an IR
  - Compile and cache the IR for subsequent executions

- **Key Advantages**
  - Users write idiomatic Python (e.g. NumPy / PyTorch)
  - Automatic specialization for shapes, dtypes, layouts
  - Easy interop with existing frameworks
  - Great for fusing high-level tensor computations

- **How It Handles Python**
  - Control flow is executed once during tracing
  - Data-dependent branches → baked into IR or trigger retrace
  - Fallback to eager for unsupported Python (graph breaks)
  - Guards ensure cached variants are reused safely

Example: Tracing-Based DSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
   for i in range(C.shape[1]):
       matrix = gemm(A, B, C, i)
   Epilogue().run(matrix)

1  kernel(Epilogue, A, B, C)
   kernel(ReLU, A, B, C)
```

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

2  @cute.jit
   def kernel(Epilogue, A, B, C):
      for i in range(C.shape[1]):
          matrix = gemm(A, B, C, i)
      Epilogue().run(matrix)

1  kernel(Epilogue, A, B, C)
   kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
```

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

2  @cute.jit
   def kernel(Epilogue, A, B, C):
3      for i in range(C.shape[1]):
           matrix = gemm(A, B, C, i)
       Epilogue().run(matrix)

1  kernel(Epilogue, A, B, C)
   kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
                                                          …
  // Begin loop
  %fA = cute.slice ...
  %fB = cute.slice ...
  %fC = cute.slice ...
  cute.gemm(%fA, %fB, %fC)
  %fA1 = cute.slice ...
  %fB1 = cute.slice ...
  %fC1 = cute.slice ...
  cute.gemm(%fA1, %fB1, %fC1)
  %fA2 = cute.slice ...
  %fB2 = cute.slice ...
  %fC2 = cute.slice ...
  cute.gemm(%fA2, %fB2, %fC2)
  … more iterations …
  // After loop
```

Where is my loop?

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))
```

2
```
@cute.jit
def kernel(Epilogue, A, B, C):
```

3
```
   for i in range(C.shape[1]):
       matrix = gemm(A, B, C, i)
```

4
```
   Epilogue().run(matrix)
```

1
```
kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
                                  …
   // Begin loop
   %fA = cute.slice ...
   %fB = cute.slice ...
   %fC = cute.slice ...
   cute.gemm(%fA, %fB, %fC)
   %fA1 = cute.slice ...
   %fB1 = cute.slice ...
   %fC1 = cute.slice ...
   cute.gemm(%fA1, %fB1, %fC1)
   %fA2 = cute.slice ...
   %fB2 = cute.slice ...
   %fC2 = cute.slice ...
   cute.gemm(%fA2, %fB2, %fC2)
   … more iterations …
   // After loop
```

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

5 2 3 4 1

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
...
  // Begin loop
  %fA = cute.slice ...
  %fB = cute.slice ...
  %fC = cute.slice ...
  cute.gemm(%fA, %fB, %fC)
  %fA1 = cute.slice ...
  %fB1 = cute.slice ...
  %fC1 = cute.slice ...
  cute.gemm(%fA1, %fB1, %fC1)
  %fA2 = cute.slice ...
  %fB2 = cute.slice ...
  %fC2 = cute.slice ...
  cute.gemm(%fA2, %fB2, %fC2)
  ... more iterations ...
  // After loop
  cute.print("identity epilogue\0A", ) :
  // We don't have fusion, so no code here
  cute.memref.store_vec %res ... : !memref_gmem_f16
```

1. Class construction worked
2. Polymorphic call worked
3. Function call is inlined.

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
   def run(self, matrix):
       print_debug("identity epilogue")
       return matrix

class ReLU(Epilogue):
   def run(self, matrix):
       print_debug("ReLU epilogue")
       cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue, A, B, C):
   for i in range(C.shape[1]):
       matrix = gemm(A, B, C, i)
   Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
1 kernel(ReLU, A, B, C)
```

Execute
2nd call

Python Interpreter

Example: Tracing-Based DSL

Fused Activation Functions

```
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) … # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix
```

5

```
class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))
```

2

```
@cute.jit
def kernel(Epilogue, A, B, C):
```

3

```
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
```

4

```
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
```

1

```
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
                                  …
    // Begin loop
    %fA = cute.slice ...
    %fB = cute.slice ...
    %fC = cute.slice ...
    cute.gemm(%fA, %fB, %fC)
    %fA1 = cute.slice ...
    %fB1 = cute.slice ...
    %fC1 = cute.slice ...
    cute.gemm(%fA1, %fB1, %fC1)
    %fA2 = cute.slice ...
    %fB2 = cute.slice ...
    %fC2 = cute.slice ...
    cute.gemm(%fA2, %fB2, %fC2)
    … more iterations …
    // After loop
    cute.print("ReLU epilogue\0A", )
    // ReLU fusion code here
    %res_relu = arith.select(%res, …)
    …
    cute.memref.store_vec %res_relu, … : !memref_gmem_f16
```

Which Tracing-Based DSL Unlock Rapid GPU Kernel Authoring?

yet hide the program's true structure

- **Advantages**
  - **Captures actual runtime behavior** — no need to re-implement Python semantics
  - **Metaprogramming works naturally** — loops, conditionals, dynamic shapes are just Python

- **Disadvantages**
  - **Control flow is opaque** — harder to analyze and transform statically
  - **Error messages are weaker a**— tied to runtime traces, not source structure

Which Python Unlocks Rapid GPU Kernel Authoring?

Our Option: **Hybrid DSL (AST + Tracing)**

- **Fact 1: Fast GPU kernels are simple at runtime**
  - No function calls
  - No object-orient programming, polymorphism
  - No complex branching or deeply fused logic

- **Fact 2: But developers love these concepts at authoring time**
  - Meta-programming
  - Clean, structured Python code

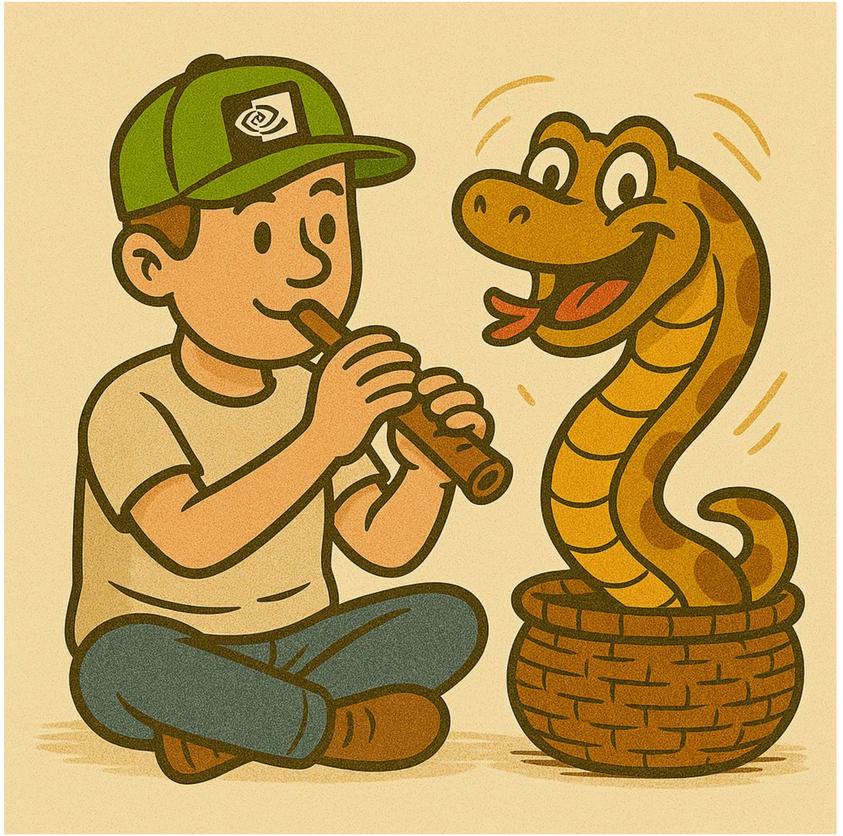
Our DSL is Hybrid: Combine AST + Tracing

When Two Worlds Collide

- **Hybrid DSL**
  - **AST** — capture program structure
  - **Tracing** — meta-programming

- **Introducing `Constexpr`**
  - Evaluated by the Python interpreter at capture time
  - Enables lightweight meta-programming without extra syntax

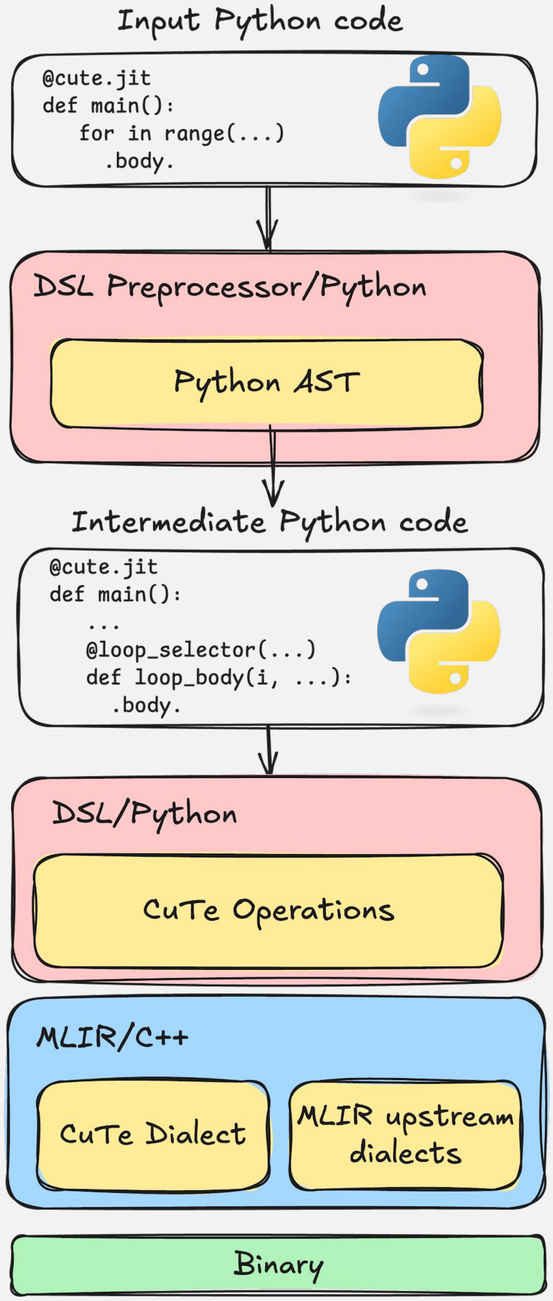
- **What can be `Constexpr`**
  - **Variables**
    - Function arguments
    - Variable definition
  - **Control flow can be `Constexpr`**
    - `for / if / else` can be resolved at compile time
  - **Shapes can be `Constexpr`**
    - User decides dynamic shapes or static shapes



Compilation flow

Meta ↔ Object: Interpreter Meets MLIR

- **Pre-Staging - Python AST**
  - Inserts callbacks to capture program structure (e.g., control flow, loops, branches).
- **Meta-Stage - Python Interpreter**
  - Executes the meta program.
  - During execution, callbacks emit MLIR IR (tracing + partial evaluation).
- **Object-Stage - MLIR Compiler**
  - MLIR IR is lowered and compiled to a target binary.
  - Runs on device (e.g., GPU/CPU/accelerator).



Understanding Stages

AST Rewrite → [print executes 🐍] → MLIR

```
$> python myprogram.py
```

```python
@cute.jit
def add_dynamicexpr(b: cutlass.Float32):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_dynamicexpr] result =", result)

@cute.jit
def add_constexpr(b: cutlass.Constexpr):
    a = 2.0
    result = a + b
    print("[add_constexpr] result =", result)

@cute.jit
def add_hybrid(b: cutlass.Constexpr):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_hybrid] result =", result)

b = 5.0
add_dynamicexpr(b)
add_constexpr(b)
add_hybrid(b)
```

Understanding Stages

AST Rewrite → [print executes 🐍] → MLIR

```
$> python myprogram.py

  [add_dynamicexpr] result = ?
```

```python
@cute.jit
def add_dynamicexpr(b: cutlass.Float32):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_dynamicexpr] result =", result)

@cute.jit
def add_constexpr(b: cutlass.Constexpr):
    a = 2.0
    result = a + b
    print("[add_constexpr] result =", result)

@cute.jit
def add_hybrid(b: cutlass.Constexpr):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_hybrid] result =", result)

b = 5.0
add_dynamicexpr(b)
add_constexpr(b)
add_hybrid(b)
```

Understanding Stages

AST Rewrite → [print executes 🐍] → MLIR

```
$> python myprogram.py

  [add_dynamicexpr] result = ?

  [add_constexpr] result = 7.0
```

```python
@cute.jit
def add_dynamicexpr(b: cutlass.Float32):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_dynamicexpr] result =", result)

@cute.jit
def add_constexpr(b: cutlass.Constexpr):
    a = 2.0
    result = a + b
    print("[add_constexpr] result =", result)

@cute.jit
def add_hybrid(b: cutlass.Constexpr):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_hybrid] result =", result)


b = 5.0
add_dynamicexpr(b)
add_constexpr(b)
add_hybrid(b)
```

Understanding Stages

AST Rewrite → [print executes 🐍] → MLIR

```
$> python myprogram.py

  [add_dynamicexpr] result = ?

  [add_constexpr] result = 7.0

  [add_hybrid] result = ?
```

```python
@cute.jit
def add_dynamicexpr(b: cutlass.Float32):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_dynamicexpr] result =", result)

@cute.jit
def add_constexpr(b: cutlass.Constexpr):
    a = 2.0
    result = a + b
    print("[add_constexpr] result =", result)

@cute.jit
def add_hybrid(b: cutlass.Constexpr):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_hybrid] result =", result)

b = 5.0
add_dynamicexpr(b)
add_constexpr(b)
add_hybrid(b)
```

Understanding Stages

AST Rewrite → [print executes 🐍] → MLIR

```
$> python myprogram.py
   [add_hybrid (meta-stage)] result = ?
   [add_hybrid (object-stage)] result =7.000000
```

```python
@cute.jit
def add_hybrid(b: cutlass.Constexpr):
    a = cutlass.Float32(2.0)
    result = a + b
    print("[add_hybrid (meta-stage)] result =", result)
    cute.printf("[add_hybrid (object-stage)] result =", result)

b = 5.0
add_hybrid(b)
```

Let's Write Our First Example with CuTeDSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue : cutlass.Constexpr,
           A : cute.Tensor, B : cute.Tensor, C : cute.Tensor):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
```

Let's Write Our First Example with CuTeDSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue : cutlass.Constexpr,
           A : cute.Tensor, B : cute.Tensor, C : cute.Tensor):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
   // Begin loop
   scf.for ... {
      %fA = cute.slice ...
      %fB = cute.slice ...
      %fC = cute.slice ...
      cute.gemm(%fA, %fB, %fC)
   }
   // After loop
```

Capture the dynamic loop

Let's Write Our First Example with CuTeDSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue : cutlass.Constexpr,
           A : cute.Tensor, B : cute.Tensor, C : cute.Tensor):
    for i in cutlass.range_constexpr(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
                                   ...
  // Begin loop
  %fA = cute.slice ...
  %fB = cute.slice ...
  %fC = cute.slice ...
  cute.gemm(%fA, %fB, %fC)
  %fA1 = cute.slice ...
  %fB1 = cute.slice ...
  %fC1 = cute.slice ...
  cute.gemm(%fA1, %fB1, %fC1)
  %fA2 = cute.slice ...
  %fB2 = cute.slice ...
  %fC2 = cute.slice ...
  cute.gemm(%fA2, %fB2, %fC2)
  ... more iterations ...
  // After loop
```

Loop is marked as constexpr, so user doesn't want to capture the control-flow. It's evaluated by the python interpreter.

Let's Write Our First Example with CuTeDSL

Fused Activation Functions

```python
print_debug = lambda message: DEBUG and cute.printf(message)
@cute.jit
def gemm(A,B,C,i) ... # implementation detail

class Epilogue:
    def run(self, matrix):
        print_debug("identity epilogue")
        return matrix

class ReLU(Epilogue):
    def run(self, matrix):
        print_debug("ReLU epilogue")
        cute.where(matrix > 0.0, matrix, cute.full_like(matrix, 0.0))

@cute.jit
def kernel(Epilogue : cutlass.Constexpr,
           A : cute.Tensor, B : cute.Tensor, C : cute.Tensor):
    for i in range(C.shape[1]):
        matrix = gemm(A, B, C, i)
    Epilogue().run(matrix)

kernel(Epilogue, A, B, C)
kernel(ReLU, A, B, C)
```

Live Generated MLIR code

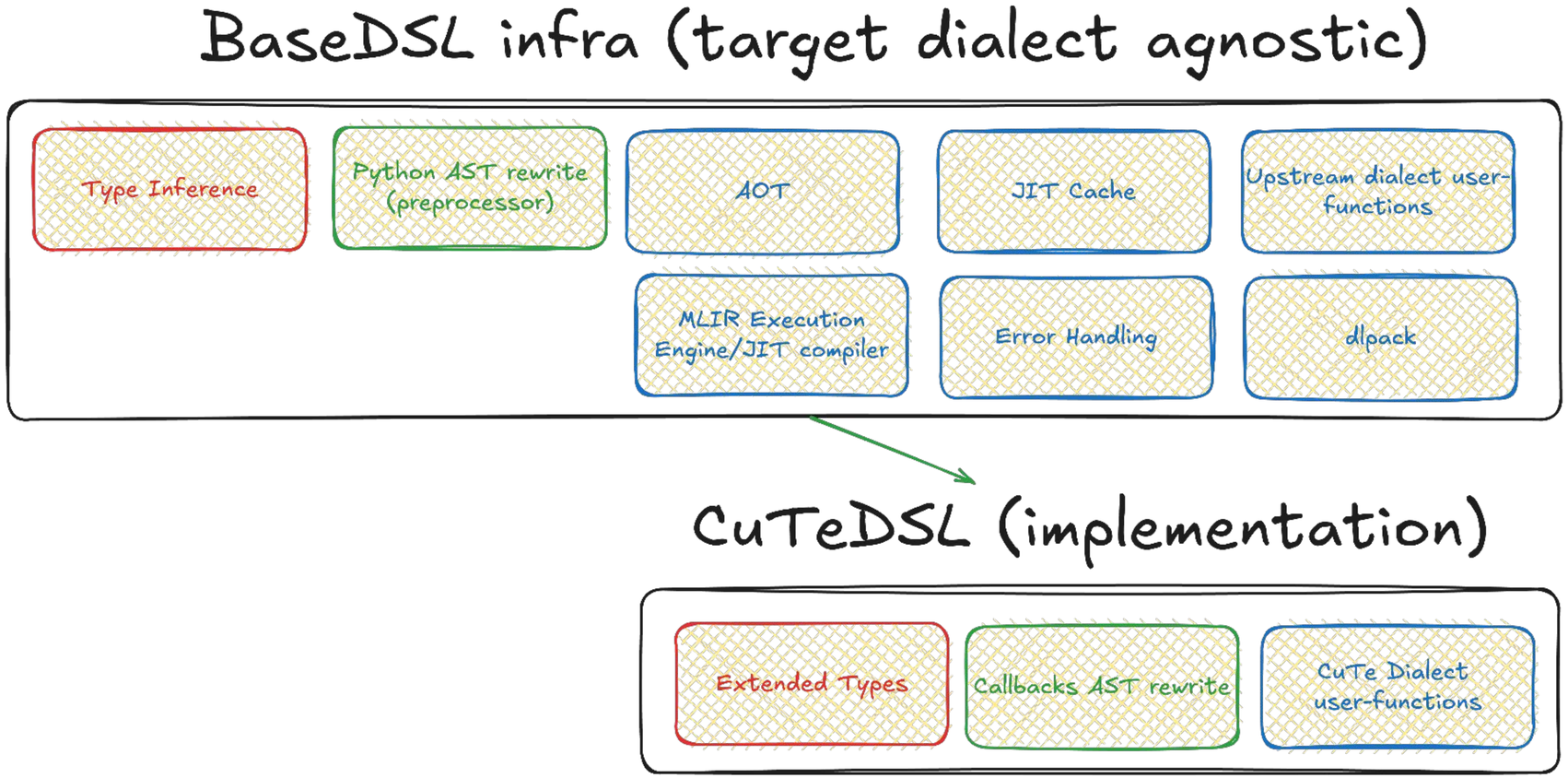
```
!memref_gmem_f16 = !cute.memref<f16, gmem, "(128,256):(256,1)">
func.func @cutlass_kernel_Epilogue(%A: !memref_gmem_f16,
                                   %B: !memref_gmem_f16,
                                   %C: !memref_gmem_f16)  {
  // Begin loop
  scf.for ... {
    %fA = cute.slice ...
    %fB = cute.slice ...
    %fC = cute.slice ...
    cute.gemm(%fA, %fB, %fC)
  }
  // After loop
```

Constexpr are evaluated in compile-time by python Interpreter

One DSL to Rule All MLIR Targets

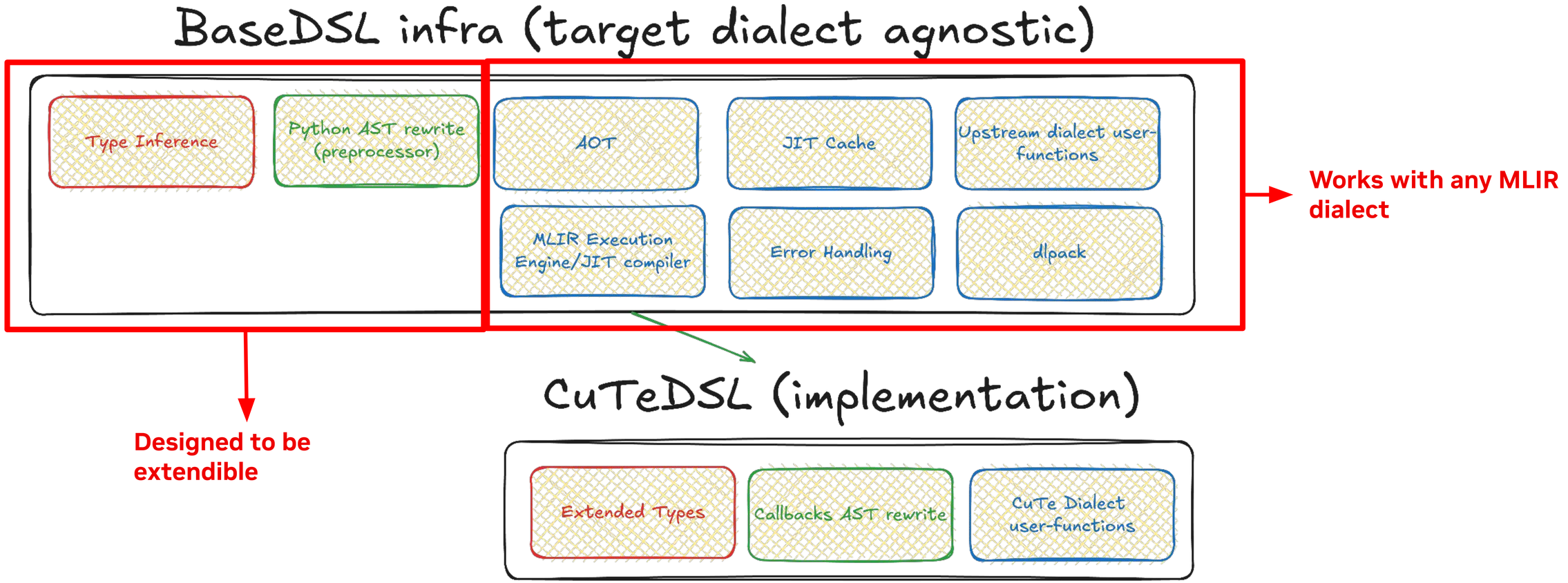
Target Dialect Agnostic Python DSL

CuTeDSL is an implementation



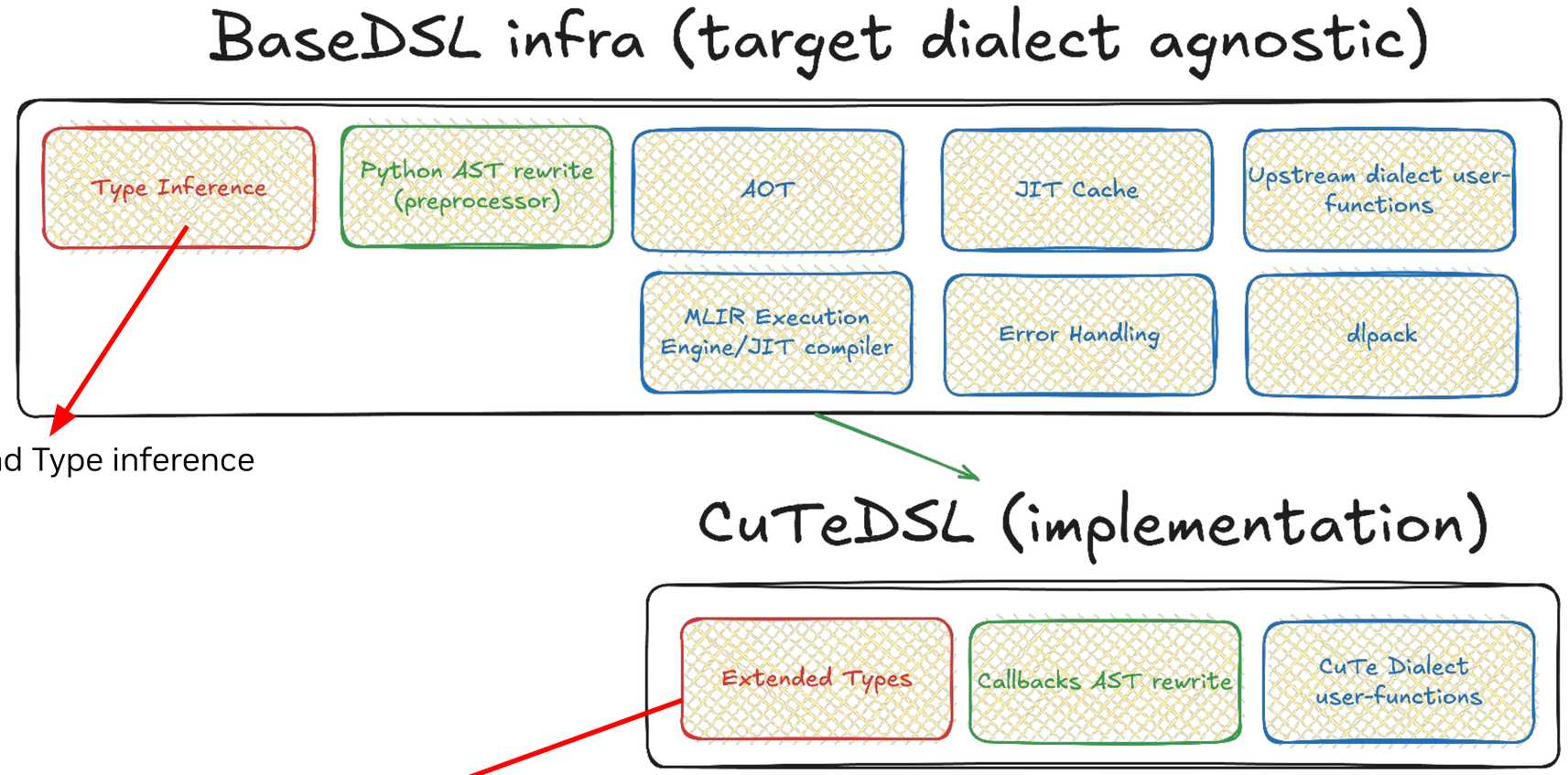
Target Dialect Agnostic Python DSL

CuTeDSL is an implementation



Target Dialect Agnostic Python DSL

CuTeDSL is an implementation



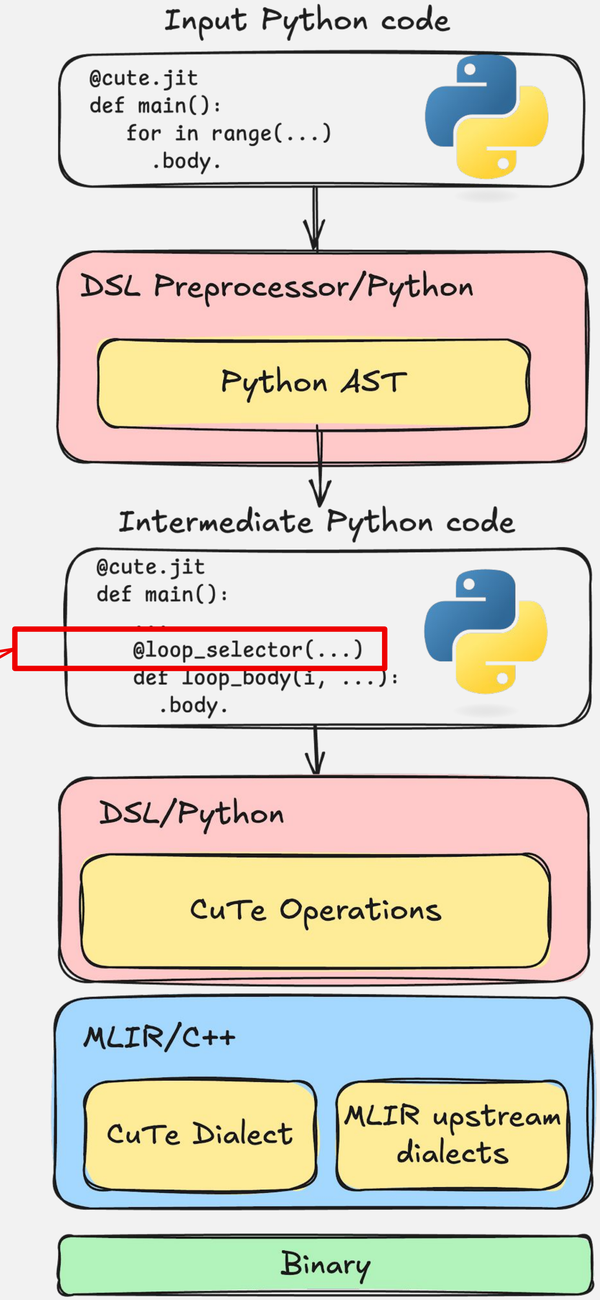
Builtin Types and Type inference

```
cutlass.Int4
cutlass.Int8
cutlass.Int16
cutlass.Int32
cutlass.Int64
cutlass.Float16
cutlass.BFloat16
cutlass.TFloat32
cutlass.Float32
cutlass.Float64
cutlass.Float8E4M3
cutlass.Float8E5M2
...
cute.Tensor
cute.CopyAtom
cute.TensorSSA
```

Staging:

Control-Flow from Python, Your Way

- **BaseDSL**
  - Rewrites Python AST → Intermediate Python
  - Hooks into callbacks for control-flow generation
- **Implementation: *CuTeDSL***
  - Plug in callbacks
  - Emit `scf/cf`
- **Implementation: *myDSL* (hypothetical)**
  - Same callbacks, different IR
  - Fully customizable control-flow



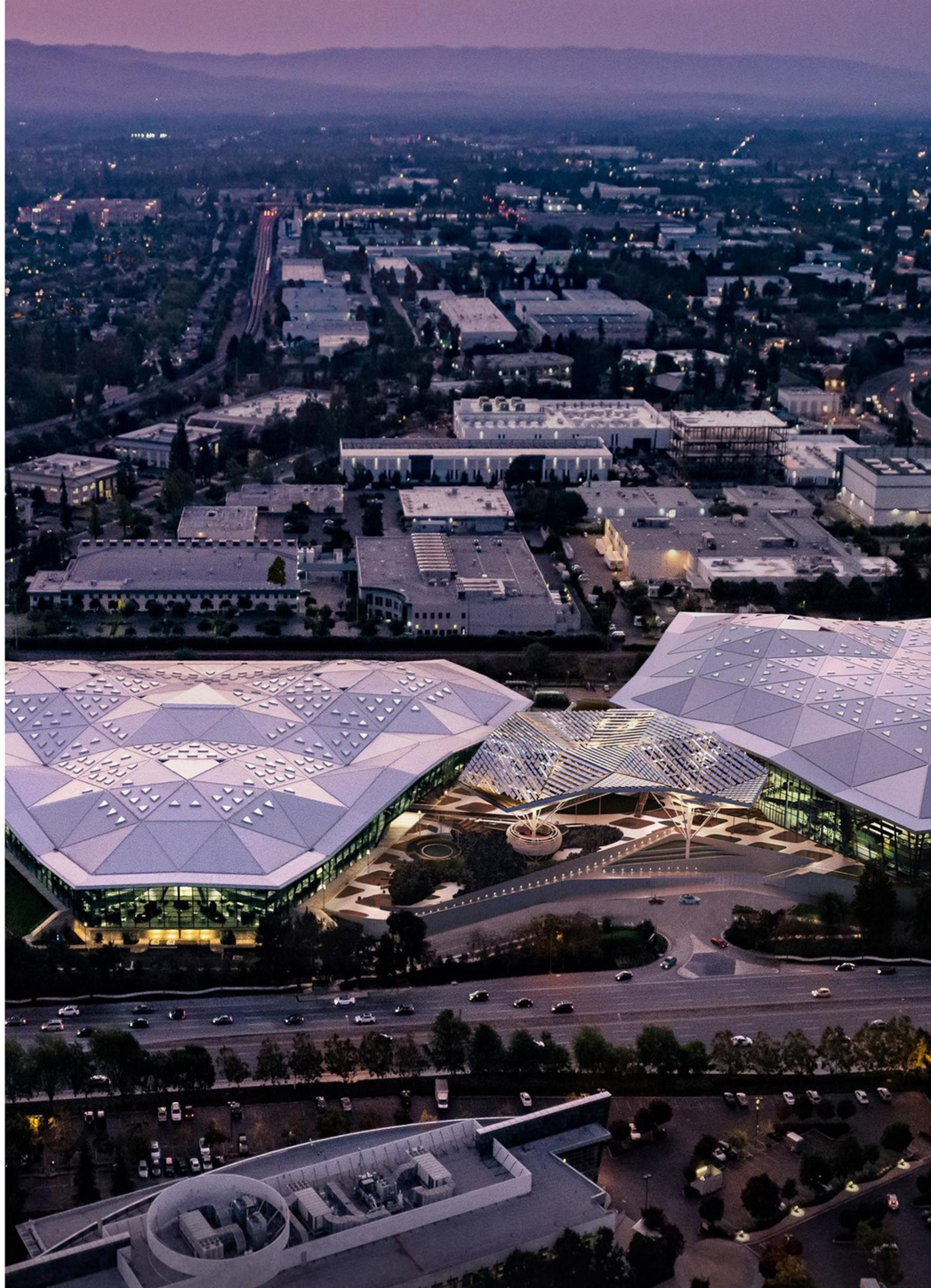
Calling Convention

How Python Talks to DSL

Caller	Callee	Allowed	Compilation/Runtime
Python function	`@jit`	✅	DSL runtime
Python function	`@kernel`	❌	N/A (error raised)
`@jit`	`@jit`	✅	Compile-time call, inlined
`@jit`	Python function	✅	Compile-time call, inlined
`@jit`	`@kernel`	✅	Dynamic call via GPU driver or runtime
`@kernel`	`@jit`	✅	Compile-time call, inlined
`@kernel`	Python function	✅	Compile-time call, inlined
`@kernel`	`@kernel`	❌	N/A (error raised)

Implemented **Cutlass Python DSL** Infrastructure

- **Target-agnostic** — works with any MLIR backend
- Three staging levels:
  - **Pre-Staging** → Python AST rewrite (structural capture)
  - **Meta-Staging** → Python interpreter (tracing & partial eval)
  - **Object-Staging** → MLIR compiler (lowering & codegen)



Help Us Shape the Future

Scan the QR code to connect with us.
NVIDIA is the engine of the world's AI infrastructure. We are the world leader in accelerated computing.

Check out our open career opportunities here:
https://www.nvidia.com/en-us/about-nvidia/careers/

Acknowledgement

A great collaboration among various teams made it possible!

- Albert Di
- Albert Xu
- Anakin Zheng
- Arvin Jou
- Brandon Sun
- Chenyang Xu
- Chunyu Wang
- Cris Cecka
- dePaul Miller
- Edward Cao
- Fung Xie
- Guray Ozen
- Hao Hu
- Hong Wang
- Jeremy Furtek
- Jie Fang
- JingZe Cui
- Kihiro Bando
- Linfeng Zheng
- Longsheng Du
- Mina Sun
- Mindy Li
- Pradeep Ramani
- Questa Wang
- Serif Yesil
- Tao Xie
- Tina Li
- Vicki Wang
- Vincent Zhang
- Vijay Thakkar
- Xiao Dong
- Xiaolei Shi
- Xinyu Wang
- Yihan Chen
- Yuhan Li
- Zekun Fan

Useful Links

CuTeDSL Homepage Introduction

CuTeDSL Jupyter Notebooks

Enable Tensor Core Programming in Python with CUTLASS 4.0

Programming Blackwell Tensor Cores with CUTLASS [GTC25]

CuTe concepts quickstart