Modular

MAX's JIT Graph Compiler

Feras Boulala
feras.boulala@modular.com
LLVM Dev Mtg 2025

Agenda

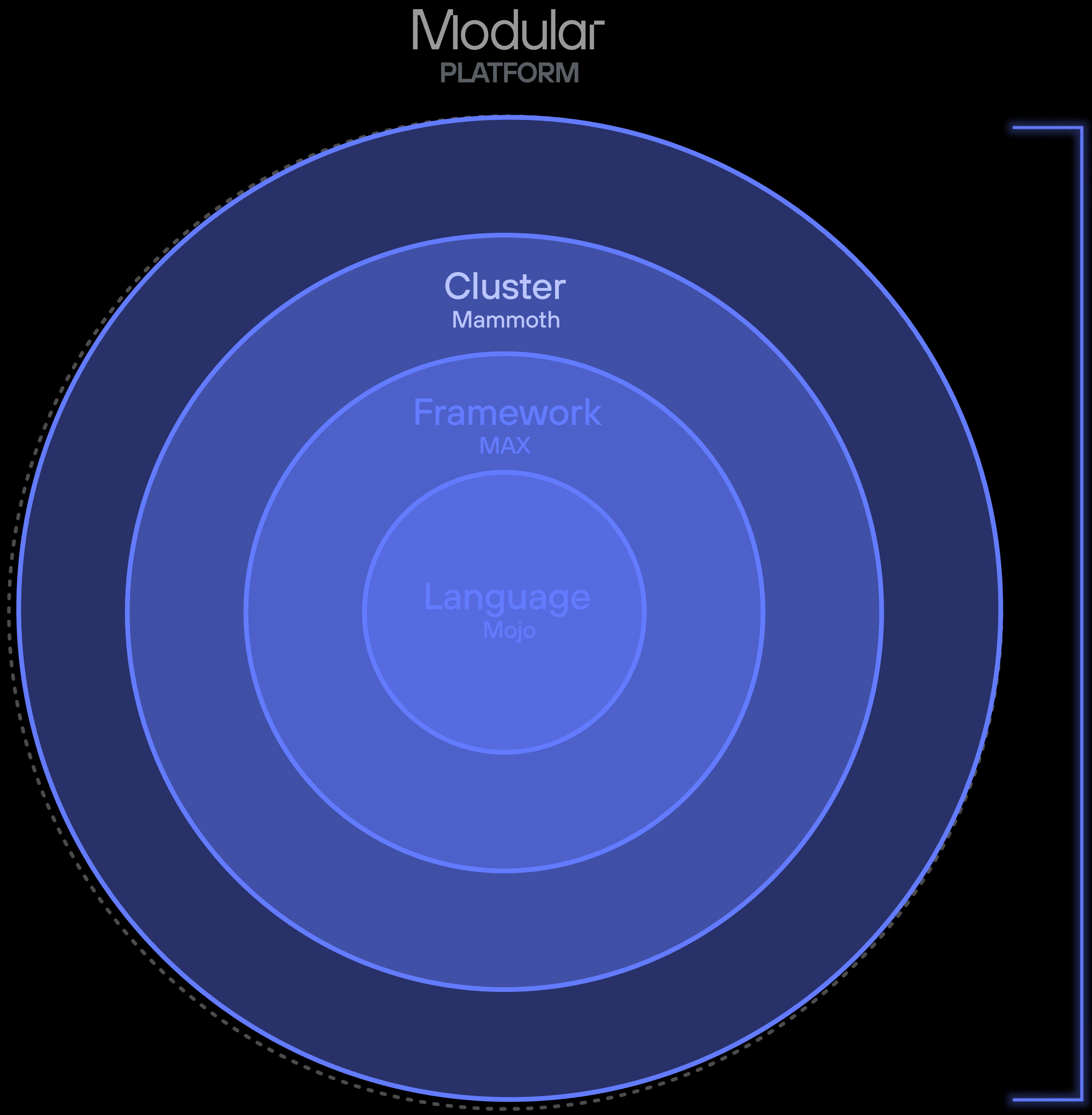
1 Graph Compiler Overview

2 Extensibility

3 MOGG

4 Mojo🔥 Codegen

5 Q&A



Modular

Inference system

Cluster

Framework

Language

Modular

Inference solution

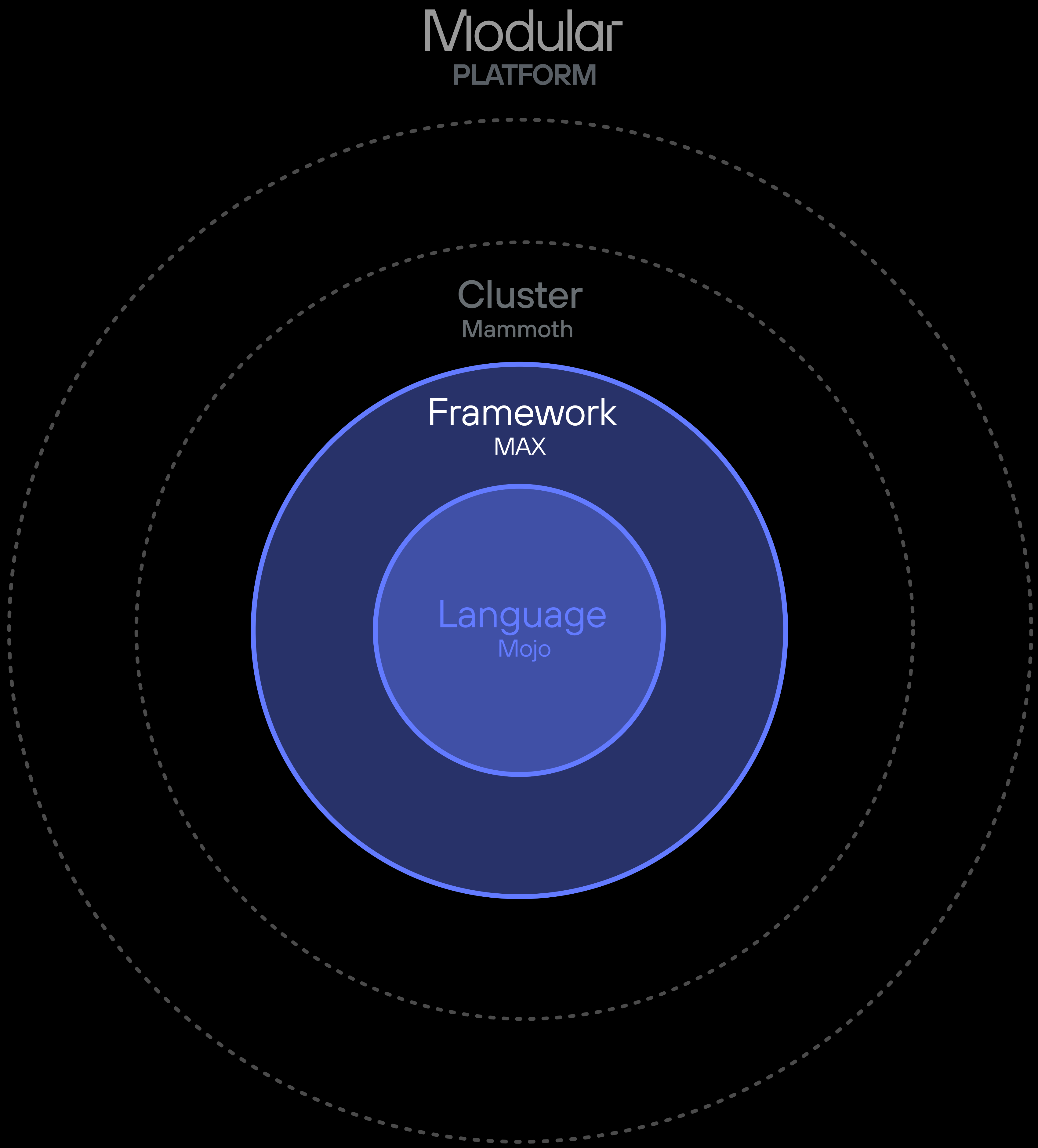
Mammoth 🦣

Deploy large-scale distributed GenAI services with SOTA perf across diverse models & hardware

KUBERNETES NATIVE

Framework

Language



Modular

Inference solution

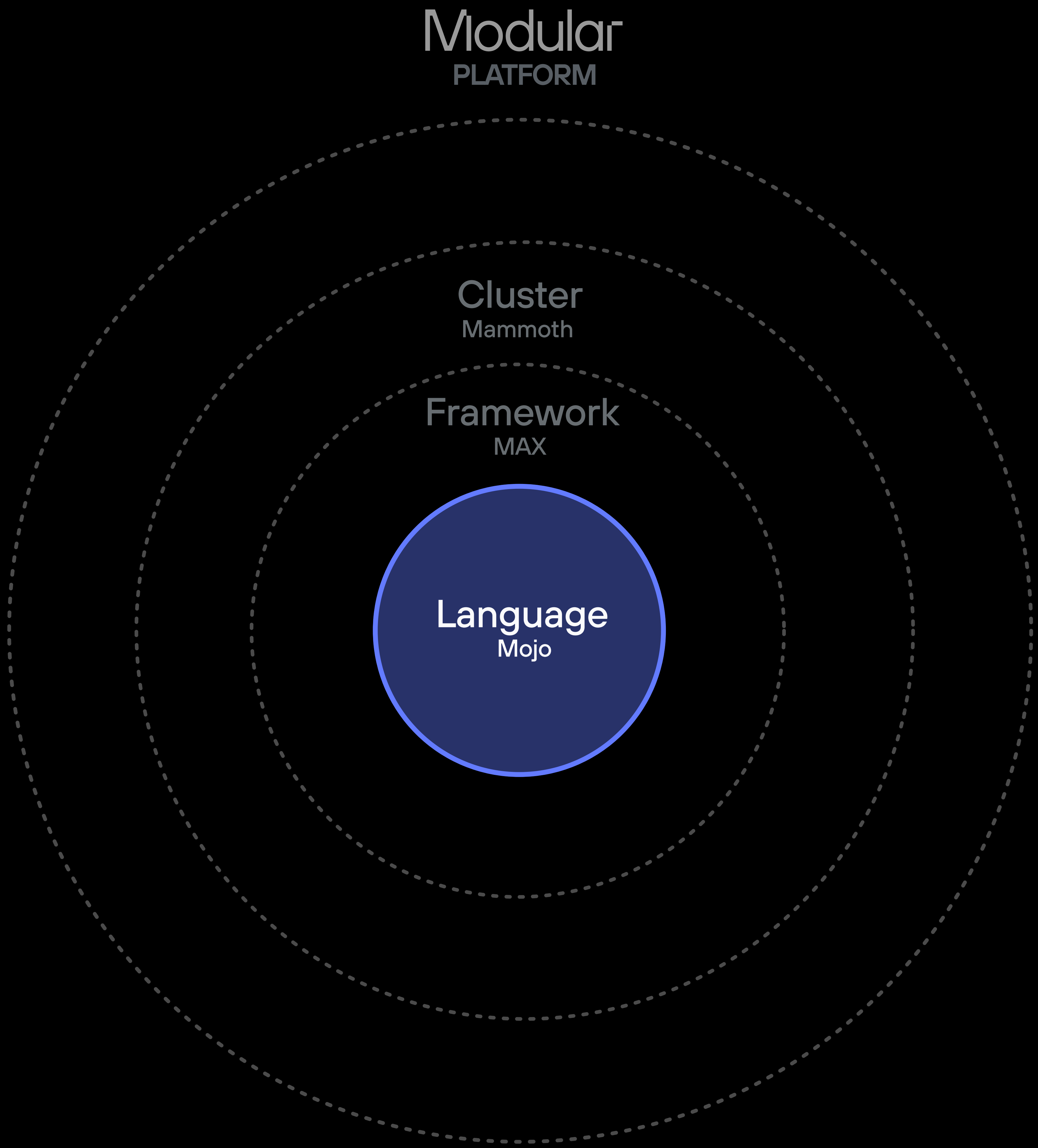
Cluster

MAX 🧑‍🚀

Serve 500+ models with SOTA Perf
Across GPU (NVIDIA, AMD) & CPU
with one container & OpenAI API

SERVE | ENGINE | KERNELS

Language



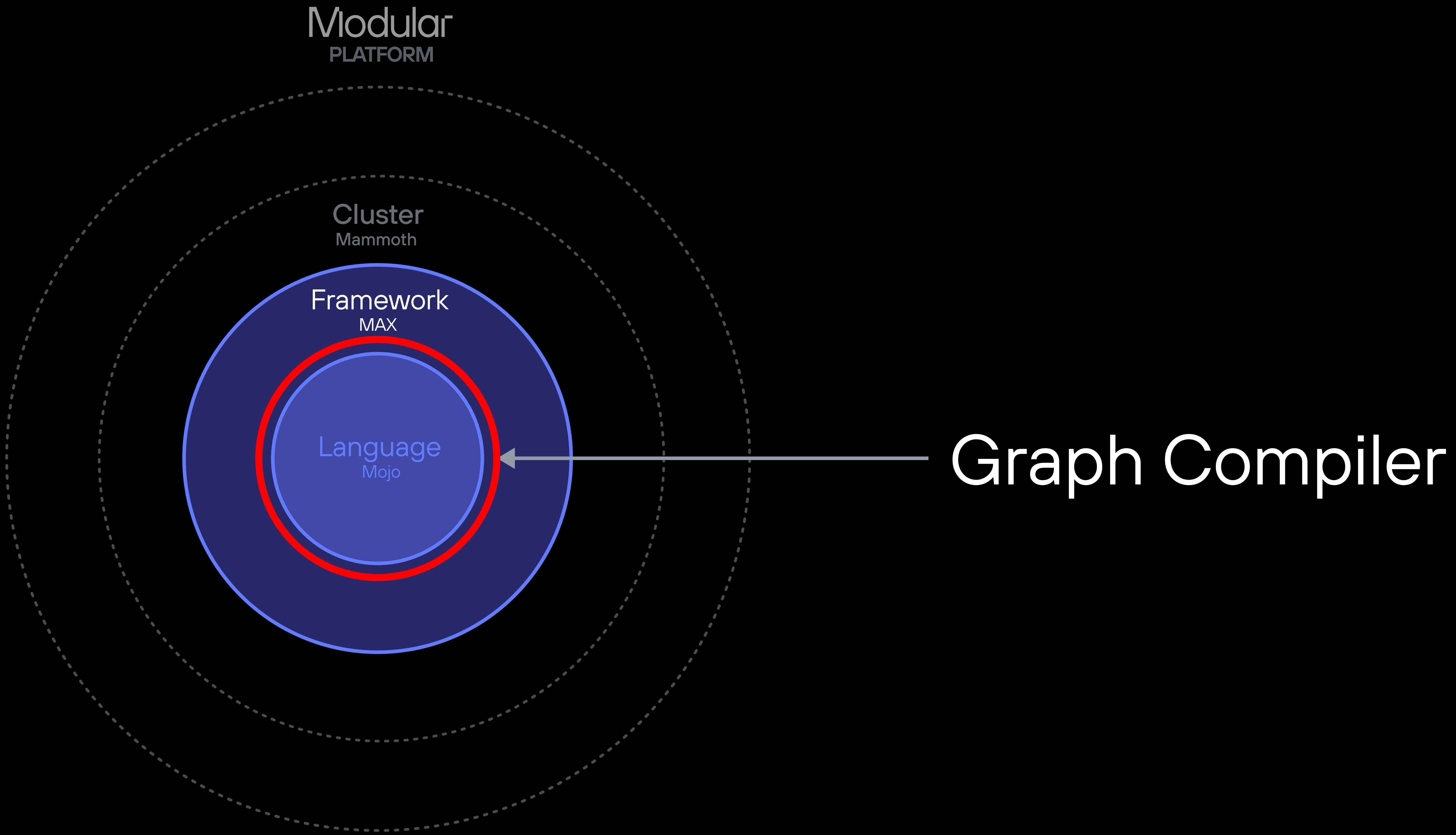
Mammoth

Framework

Mojo 🔥

Invent novel AI algorithms with one, pythonic systems language that runs across any AI hardware with cutting-edge tooling

STD LIB | COMPILER | TOOLS



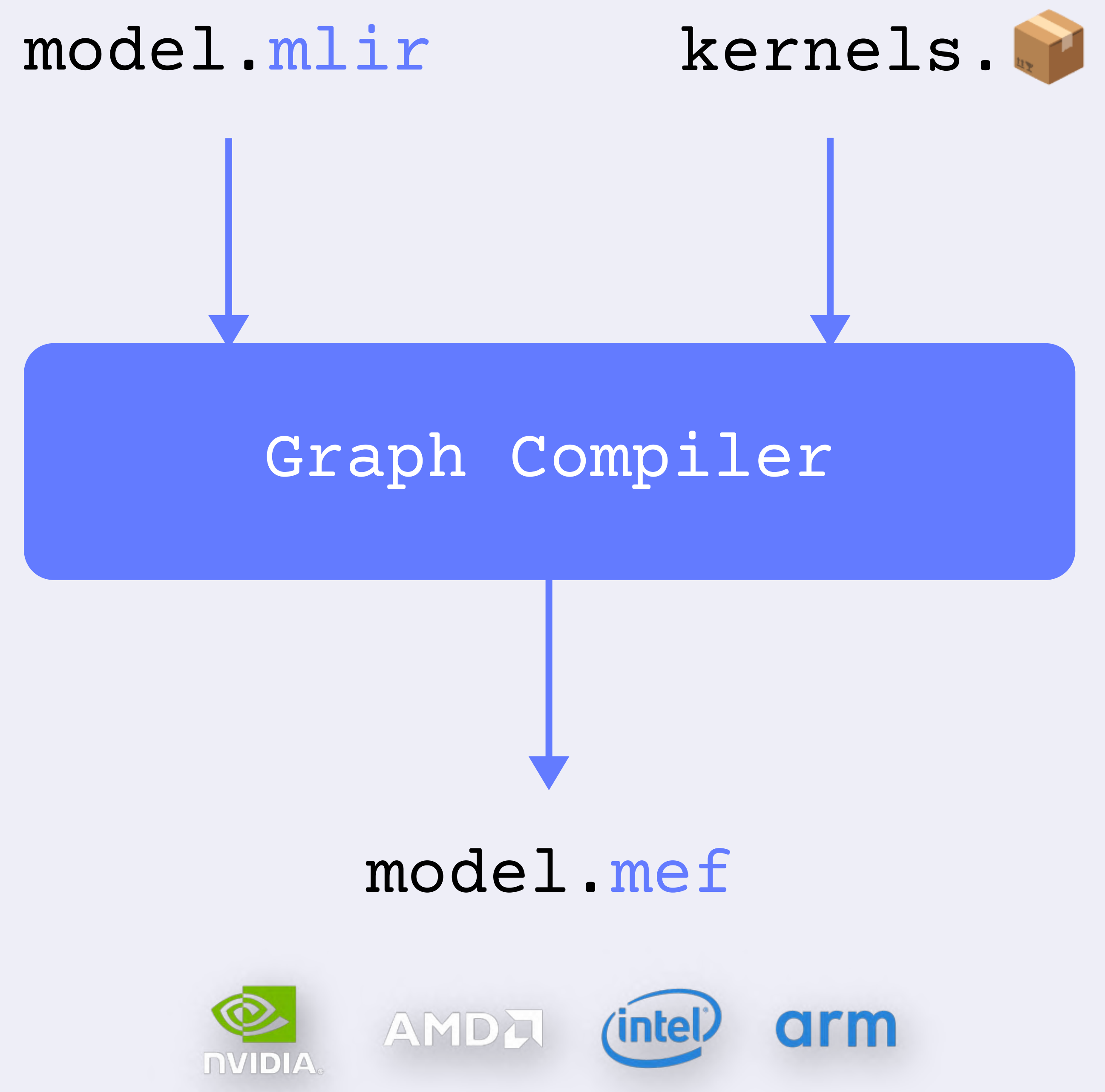
Modular
PLATFORM
Cluster
Mammoth
Framework
MAX
Language
Mojo
Graph Compiler

Graph Compiler Overview

Inputs and outputs

RMO/MO
MOGG
MGP

- Inputs:
  - MLIR DAG of DL ops (built using a Python Graph API)
  - Package kernels
- Output: DAG of executable kernels in the **M**odular **E**xecutable **F**ormat (MEF).



Graph Compiler Overview

Inputs and outputs
RMO/MO
MOGG
MGP

- (**R**elaxed) **M**odular **O**perators
- RISC-like, only 135 operators
- Vertices = operators, edges = tensors
- Side effects through chains and mutable tensors
- Optimizations:
    - Symbolic folding
    - Shape inference
    - Constant folding

```
mo.graph @model<N, M, K>(
 %A: !mo.tensor<[M, K], f32>,
 %B: !mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>,
 %bias: !mo.tensor<[], f32>
) -> (!mo.tensor<[M, N], f32>) {

 %C = mo.matmul(%A, %B) : (
   !mo.tensor<[M, K], f32>,
   !mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>
 )  -> !mo.tensor<[M, N], f32>

 %bcast_bias = mo.static.broadcast_to(%bias)
   : !mo.tensor<[], f32> -> !mo.tensor<[M, N], f32>

 %add = mo.add(%C, %bcast_bias) : !mo.tensor<[M, N], f32>
 %out = mo.relu(%add) : !mo.tensor<[M, N], f32>

 mo.output %out : !mo.tensor<[M, N], f32>
}
```

Graph Compiler Overview

Inputs and outputs

RMO/MO

MOGG

MGP

- **MO**dular **G**raph **G**enerators
- Fusion dialect
- Structured kernels that compose via lambdas
- Destination passing style (bufferized) semantics
- Optimizations:
  - elementwise fusion
  - prologue and epilogue fusion
  - view fusions
  - small constant inlining

```
  %0 = mogg.experimental.kernel(
    %arg0: !mo.tensor<[M, K], f32>,
    %arg1: !mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>
  ) -> !mo.tensor<[M, N], f32> {

    %4 = mogg.output_placeholders : !mo.tensor<[M, N], f32>

    %5 = mogg.bind (%4) output_lambda (%arg5: f32, %arg6:
!pop.array<2, index>) {
      mogg.output %arg5, %arg6 : f32, !pop.array<2, index>
    }: (!mo.tensor<[M, N], f32>) -> !mo.tensor<[M, N], f32>

    %6 = mogg.device_context_placeholder {device =
#M.device_ref<"cpu", 0>} : !mogg.context

    mogg.call.execute["mo.matmul"]
      inputs(%arg0, %arg1, %6
        : !mo.tensor<[M, K], f32>, !mo.tensor<[K, N], f32,
{layout = #mo.layout<KN>}>, !mogg.context)
      outputs(%5
        : !mo.tensor<[M, N], f32>) {kernel_param = {packed_b =
false, transpose_b = false}}

  } {allocIndices = [], device = #M.device_ref<"cpu", 0>,
hasReadEffect = false, hasWriteEffect = false,
```

Graph Compiler Overview

Inputs and outputs

RMO/MO

MOGG

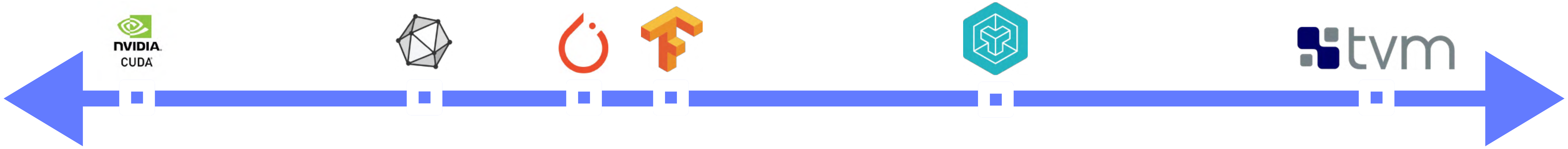
MGP

- **M**odular **G**lue **P**rimitives
- Runtime primitives dialect
- Ends in a JIT compilation (kernels aren't precompiled, only packaged)
- Optimizations:
  - sequence fusion (kernel inlining),
  - memory planning
  - execution invariant code motion ("runtime constant folding")

```
 mgp.model @model : mof {argument_device_indices = [0 : ui32,
0 : ui32, 0 : ui32], result_device_indices = [0 : ui32]}
devices [#M.device_ref<"cpu", 0>] init (%arg0:
!mgp.opaque<"weights_registry">, %arg1:
!mgp.device_context<<"cpu", 0>>) {
   %0 = grt.chain.create ()
   %1 = mgp.context.create 1
   %2 = mgp.runtime.create %0, %1[0]
   mef.output %2, %1, %arg0, %arg1 : !mef.chain,
!mgp.context, !mgp.opaque<"weights_registry">,
!mgp.device_context<<"cpu", 0>>
 } execute (%arg0: !mef.chain, %arg1: !mgp.context, %arg2:
!mgp.opaque<"weights_registry">, %arg3:
!mgp.device_context<<"cpu", 0>>, %arg4: !mgp.tensor<?x?xf32>,
%arg5: !mgp.tensor<?x?xf32>, %arg6: !mgp.tensor<f32>) ->
(!mef.chain, !mgp.tensor<?x?xf32>) {
   %idx4 = index.constant 4
   %0 = mgp.tensor.extract.tensor_spec %arg4 : <?x?xf32>
   %1 = mgp.tensor_spec.get_dim[0] %0 : <?x?xf32>
   %2 = mgp.tensor_spec.get_dim[1] %0 : <?x?xf32>
   %3 = index.casts %1 : index to si64
   %4 = index.casts %2 : index to si64
   %5 = mgp.tensor.extract.tensor_spec %arg5 : <?x?xf32>
   %6 = mgp.tensor_spec.get_dim[0] %5 : <?x?xf32>
   %7 = mgp.tensor_spec.get_dim[1] %5 : <?x?xf32>
   %8 = index.casts %6 : index to si64
   %9 = pop.cast_from_builtin %4 : si64 to !pop.scalar<si64>
   . . .
```

Modeling DL optimizing compilers

1D spectrum on this slide, but should really be a hypercube of tradeoffs 🧊



Kernels centric

- Speed of light perf
- Fast to compile
- Predictable behaviour
- Fully extensible
- Fully debuggable
- Low coverage
- Not scalable

Compiler centric

- High coverage
- Productive abstractions
- Subpar performance
- Sometimes slow compile times
- Unpredictable behaviour
- Extensibility restricted by programming model
- Indirect debugging

No finite opset, DSL or programming model can reliably cater to all of the DL infra needs

How do you design the compiler to to support current needs and make it resilient to rapid change?

Hedge by investing in programmability and extensibility

No finite opset, DSL or programming model can reliably cater to all of the DL infra needs

How do you design the compiler to to support current needs and make it resilient to rapid change?

Hedge by investing in programmability and extensibility

No finite opset, DSL or programming model can reliably cater to all of the DL infra needs

How do you design the compiler to to support current needs and make it resilient to rapid change?

Hedge by investing in programmability and extensibility

Extensibility and programmability

01

Productive kernel writing

Kernel writing should be a productive endeavour. Mojo 🔥 makes it easier to write and to compose things into libraries.

02

GC acts as kernel infra

The graph compiler should support kernel writing with compile time reflection and productive features.

03

Get out of the way of experts

When systems evolve or programming models do not fit, provide a side channel that allows complete low level control.

Extensibility and programmability

OCTOBER 17, 2025

Achieving State-of-the-Art Performance on AMD MI355 — in Just 14 Days

TRACY SHARPE · ANAND PRATAP SINGH · PRINCE JAIN · ABDUL DAKKAK · ENGINEERING

01

Productive kernel writing

Kernel writing should be a productive endeavour. Mojo 🔥 makes it easier to write and to compose things into libraries.

02

GC acts as kernel infra

The graph compiler should support kernel writing with compile time reflection and productive features.

03

Get out of the way of experts

When systems evolve or programming models do not fit, provide a side channel that allows complete low level control.

Extensibility and programmability

01

Productive kernel writing

Kernel writing should be a productive endeavour. Mojo 🔥 makes it easier to write and to compose things into libraries.

02

GC acts as kernel infra

The graph compiler should support kernel writing with compile time reflection and productive features.

03

Get out of the way of experts

When systems evolve or programming models do not fit, provide a side channel that allows complete low level control.

Extensibility and programmability

01

Productive kernel writing

Kernel writing should be a productive endeavour. Mojo 🔥 makes it easier to write and to compose things into libraries.

02

GC acts as kernel infra

The graph compiler should support kernel writing with compile time reflection and productive features.

03

Get out of the way of experts

When systems evolve or programming models do not fit, provide a side channel that allows complete low level control.

Extensibility and programmability

Validate ideas in Mojo

implement in the graph compiler if needed

01

Productive kernel writing

Kernel writing should be a productive endeavour. Mojo 🔥 makes it easier to write and to compose things into libraries.

02

GC acts as kernel infra

The graph compiler should support kernel writing with compile time reflection and productive features.

03

Get out of the way of perf

When systems evolve or programming models do not fit, provide a side channel that allows complete low level control.

Extensibility

mo.custom

mo.opaque<T>

No rebuilds

- Generic operator for arbitrary custom kernels
- Operation semantics are unknown to the GC
- Side effects handled like every other native operator (chains and buffers)
- Attributes lower to kernel parameters

```
def MO_CustomOp : MO_TensorOp<"custom", [
 DeclareOpInterfaceMethods<MO_DefaultParameterization>,
 DeclareOpInterfaceMethods<MO_MutableOpInterface>,
 DeclareOpInterfaceMethods<MO_ConditionallyInPlaceInterface>,
 DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
 MO_ExplicitDevice,
]> {
 let arguments = (ins Variadic<AnyType>:$operands,
                      StrAttr:$symbol,
                      DefaultValuedAttr<M_DeviceRefAttr,
"\"cpu\", 0">:$device,
                      DefaultValuedAttr<DictionaryAttr, "
{}">:$parameters,
                      MO_ParamDecls:$outputParamDecls);
 let results = (outs Variadic<AnyType>:$results);
```

Extensibility

mo.custom
mo.opaque<T>
No rebuilds

- Generic MLIR type
- String attribute represents Mojo type
- Attribute dictionary that reflects Mojo parameters on the type
- Fully opaque from the PoV of the GC (hence the name)

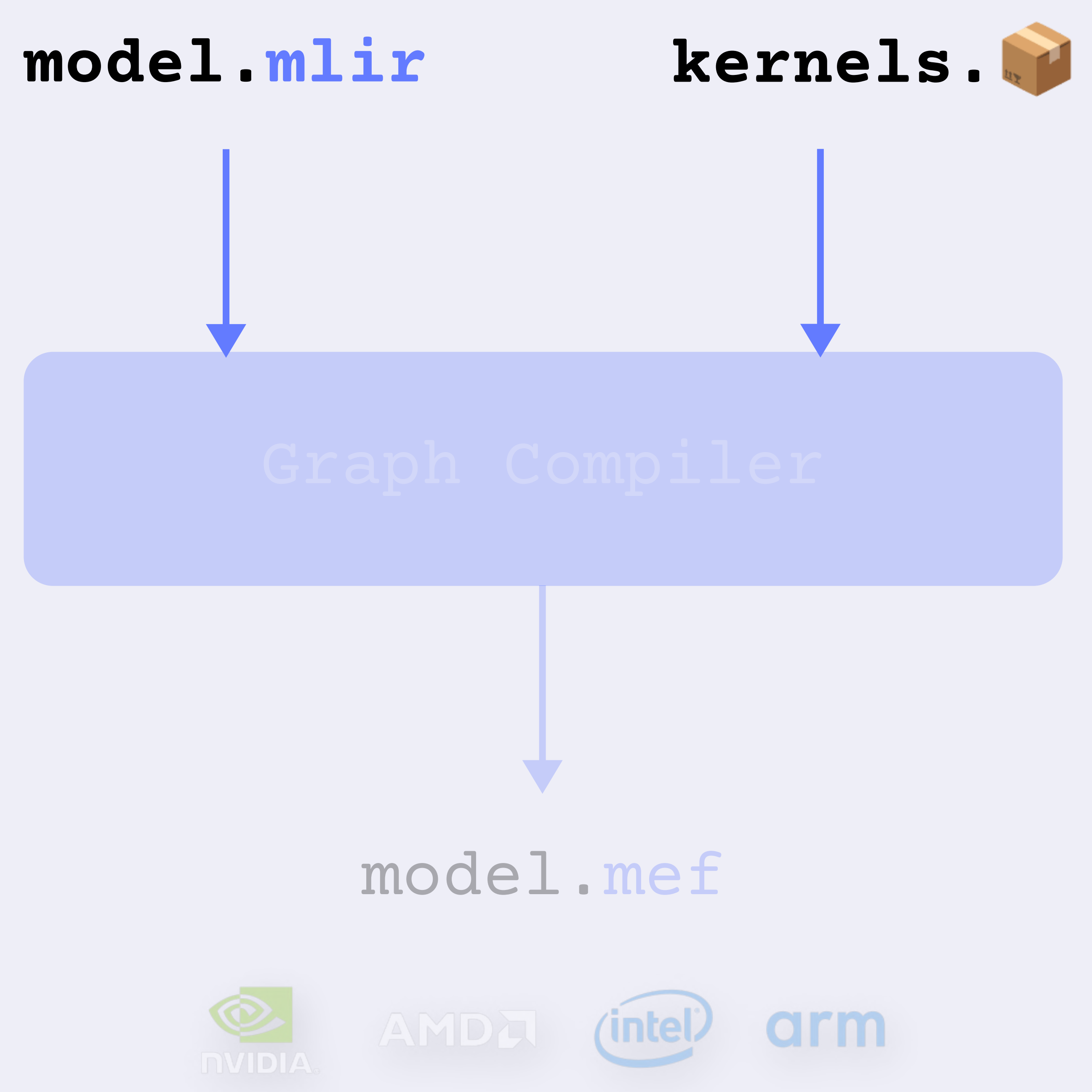
```
def MO_Opaque : MO_Type<"Opaque", "opaque"> {
 let summary = "Opaque MO type.";
 let description = [{
   This is a custom user-defined type.
     Example:
     ```mlir
       !mo.opaque<"my_list">
       !mo.opaque<"my_list", {foo = 42}>
     ```
 }];

 let parameters = (ins
   "StringAttr": $symbol,
   DefaultValuedParameter<"::mlir::DictionaryAttr",
"::mlir::DictionaryAttr::get($_ctxt)">:$parameters
 );
```

Extensibility

mo.custom
mo.opaque<T>
No rebuilds

- The GC does not require a rebuild.
- Simply provide the MLIR model graph with custom operations and types and packaged kernels
- **JIT** will take care of the actual compilation
- Compile time is reduced via various levels of caches (model-level, GC IR and Mojo IR levels)



How do you bridge the gap?

```
@compiler.register("mo.matmul")
struct Matmul:
  @staticmethod
  fn execute[
      transpose_b: Bool,
      packed_b: Bool,
      lambdas_have_fusion: Bool,
      target: StaticString,
      _trace_name: StaticString,
  ](
      c: _FusedComputeOutputTensor[rank=2],
      a: InputTensor[rank=2],
      b: InputTensor[rank=2],
      ctx: DeviceContextPtr,
  ) capturing raises:
...
```

Mojo

```
  %0 = mogg.experimental.kernel(
    %arg0: !mo.tensor<[M, K], f32>,
    %arg1: !mo.tensor<[K, N], f32, {layout =
#mo.layout<KN>}>
  ) -> !mo.tensor<[M, N], f32> {

    %4 = mogg.output_placeholders : !mo.tensor<[M, N], f32>

    %5 = mogg.bind (%4) output_lambda (%arg5: f32, %arg6:
!pop.array<2, index>) {
      mogg.output %arg5, %arg6 : f32, !pop.array<2, index>
    }: (!mo.tensor<[M, N], f32>) -> !mo.tensor<[M, N], f32>

    %6 = mogg.device_context_placeholder {device =
#M.device_ref<"cpu", 0>} : !mogg.context

    mogg.call.execute["mo.matmul"]
      inputs(%arg0, %arg1, %6
        : !mo.tensor<[M, K], f32>, !mo.tensor<[K, N], f32,
{layout = #mo.layout<KN>}>, !mogg.context)
      outputs(%5
        : !mo.tensor<[M, N], f32>) {kernel_param =
{packed_b = false, transpose_b = false}}

  } {allocIndices = [], device = #M.device_ref<"cpu", 0>,
hasReadEffect = false, hasWriteEffect = false,
```

Structured MLIR kernel

MOGG

Mojo🔥 introspection
Structured kernels
Fusion

- Mojo parses directly to an MLIR dialect: LIT
- LIT fully models the language
- LIT is "pre-elaboration" meaning parameters are not yet bound to a value
- Ideal for introspecting static information (registration name, fusion opt-in, input and output identification, type of kernel like elementwise, view or normal)
- Define primitives in Mojo → Introspect them

```
@compiler.register("mo.add")
struct Add(ElementwiseBinaryOp):
    @staticmethod
    fn elementwise[
        dtype: DType,
        width: Int,
    ](lhs: SIMD[dtype, width], rhs:
SIMD[dtype, width]) -> SIMD[dtype, width]:
        return lhs + rhs
```

```
   lit.struct.decl
@Add(!AnyType_UnknownDestructibility_ElementwiseBinaryOp)
attributes {sourceName = #Add_name}
     decorators <... :string "mo.add">, *?))> {
     lit.fn @"elementwise[::DType,::Int](::SIMD[$0,
$1],::SIMD[$0, $1])"<dtype: !DType, width: !Int>(%lhs:
!lit.struct<#SIMD <:!DType dtype, :!Int width>>, %rhs:
!lit.struct<#SIMD <:!DType dtype, :!Int width>>) ->
!lit.struct<#SIMD <:!DType dtype, :!Int width>> attributes
{isStatic, sourceName = "elementwise", specialFnKind = 0 : i8}
{
       %0 = lit.call @stdlib::...::@SIMD::@"__add__..."<:!DType
 dtype, :!Int width>(%lhs, %rhs) : !lit.generator<("self":
...)>
       lit.return %0 : !lit.struct<#SIMD <:!DType dtype, :!Int
width>>
       lit.end_fn
     }
```

MOGG

Mojo🔥 introspection
Structured kernels
Fusion

- Given statically introspected information, we can construct structured MLIR kernels that mirror the Mojo kernels
- All the GC needs is the signature of the kernel → Kernel internals are currently opaque to the GC

```
%0 = mogg.experimental.kernel(
  %arg0: !mo.tensor<[M, K], f32>,
  %arg1: !mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>
) -> !mo.tensor<[M, N], f32> {

  %4 = mogg.output_placeholders : !mo.tensor<[M, N], f32>

  %5 = mogg.bind (%4) output_lambda (%arg5: f32, %arg6:
!pop.array<2, index>) {
    mogg.output %arg5, %arg6 : f32, !pop.array<2, index>
  }: (!mo.tensor<[M, N], f32>) -> !mo.tensor<[M, N], f32>

  %6 = mogg.device_context_placeholder {device =
#M.device_ref<"cpu", 0>} : !mogg.context

  mogg.call.execute["mo.matmul"]
    inputs(%arg0, %arg1, %6
      : !mo.tensor<[M, K], f32>, !mo.tensor<[K, N], f32,
{layout = #mo.layout<KN>}>, !mogg.context)
    outputs(%5
      : !mo.tensor<[M, N], f32>) {kernel_param = {packed_b =
false, transpose_b = false}}

} {allocIndices = [], device = #M.device_ref<"cpu", 0>,
hasReadEffect = false, hasWriteEffect = false,
```

MOGG

Mojo🔥 introspection
Structured kernels
Fusion

- Lambda based fusions that make the kernel capture arbitrary elementwise subgraphs:
  - Elementwise
  - Prologue
  - Epilogue
- View fusion based on tensor stride modifications
- Small constant inlining
- In-place mutation optimization

```
%C = mo.matmul(%A, %B) : (
  !mo.tensor<[M, K], f32>,
  !mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>
)  -> !mo.tensor<[M, N], f32>
%bcast_bias = mo.static.broadcast_to(%bias)
  : !mo.tensor<[], f32> -> !mo.tensor<[M, N], f32>
%add = mo.add(%C, %bcast_bias) : !mo.tensor<[M, N], f32>
%out = mo.relu(%add) : !mo.tensor<[M, N], f32>
```

```
%0 = mogg.experimental.kernel(%arg2: !mo.tensor<[], f32>,
%arg0: !mo.tensor<[M, K], f32>, %arg1: !mo.tensor<[K, N], f32,
{layout = #mo.layout<KN>}>) -> !mo.tensor<[M, N], f32> {
   %1 = mogg.output_placeholders : !mo.tensor<[M, N], f32>
   %2 = mogg.tensor.shape<!mo.tensor<[M, N], f32>>
   %3 = mogg.call.view["mo.static.broadcast_to"] (%arg2, %2)
{kernel_param = {}} : !mo.tensor<[], f32>, !mogg.shape ->
!mo.tensor<[M, N], f32>
   %4 = mogg.bind (%1) output_lambda (%arg6: f32, %arg7:
!pop.array<2, index>) {
     %6 = "mogg.tensor.load"(%3, %arg7) : (!mo.tensor<[M, N],
f32>, !pop.array<2, index>) -> f32
     %7 = mogg.call.elementwise["mo.add"] (%arg6, %6) : f32,
f32 -> f32
     %8 = mogg.call.elementwise["mo.relu"] (%7) : f32 -> f32
     mogg.output %8, %arg7 : f32, !pop.array<2, index>
   }: (!mo.tensor<[M, N], f32>) -> !mo.tensor<[M, N], f32>
   %5 = mogg.device_context_placeholder {device =
#M.device_ref<"cpu", 0>} : !mogg.context
   mogg.call.execute["mo.matmul"] ...
}
```

No difference between MO-native operators and custom ops

Fusion enabled via the kernel signature, not the op's semantics

Mojo🔥 codegen (WIP)

```
%0 = mogg.experimental.kernel(%arg2: !mo.tensor<[],
f32>, %arg0: !mo.tensor<[M, K], f32>, %arg1:
!mo.tensor<[K, N], f32, {layout = #mo.layout<KN>}>) ->
!mo.tensor<[M, N], f32> {
   %1 = mogg.output_placeholders : !mo.tensor<[M, N],
f32>
   %2 = mogg.tensor.shape<!mo.tensor<[M, N], f32>>
   %3 = mogg.call.view["mo.static.broadcast_to"]
(%arg2, %2) {kernel_param = {}} : !mo.tensor<[], f32>,
!mogg.shape -> !mo.tensor<[M, N], f32>
   %4 = mogg.bind (%1) output_lambda (%arg6: f32,
%arg7: !pop.array<2, index>) {
     %6 = "mogg.tensor.load"(%3, %arg7) :
(!mo.tensor<[M, N], f32>, !pop.array<2, index>) -> f32
     %7 = mogg.call.elementwise["mo.add"] (%arg6, %6)
: f32, f32 -> f32
     %8 = mogg.call.elementwise["mo.relu"] (%7) : f32 -
> f32
     mogg.output %8, %arg7 : f32, !pop.array<2, index>
   }: (!mo.tensor<[M, N], f32>) -> !mo.tensor<[M, N],
f32>
   %5 = mogg.device_context_placeholder {device =
#M.device_ref<"cpu", 0>} : !mogg.context
   mogg.call.execute["mo.matmul"] ...
 }
```

Fused MLIR kernel

```
fn stub_0(
    arg_0: ManagedTensorSlice[...],
    arg_1: ManagedTensorSlice[...],
    arg_2: ManagedTensorSlice[...],
    arg_3: ManagedTensorSlice[...],
    arg_4: asyncrt.DeviceContextPtr
) raises:
   var var_5 = rebind[InputTensor[...]](arg_0)
   alias param_6 = DimList.create_unknown[2]()
   var var_7 = StaticBroadcastTo.update_input_view[...](var_5,
arg_3.shape())
   var var_9 = rebind[InputTensor[...]](arg_3)
   @parameter
   @always_inline
   fn output_lambda_10[...](
        indices_12: IndexList[2],
        input_13: SIMD[...]
   ):
       var var_14 = simd_load_from_managed_tensor_slice[...](var_7,
indices_12)
       var var_15 = Add.elementwise[...](input_13, var_14)
       var var_16 = ReLU.elementwise[...](var_15)
       simd_store_into_managed_tensor_slice[...](var_9, indices_12,
var_16)

   var var_17 = rebind[_FusedComputeOutputTensor[...](arg_3)
   var var_18 = rebind[InputTensor[...]](arg_1)
   var var_19 = rebind[InputTensor[...]](arg_2)
   Matmul.execute[transpose_b = False, packed_b = False, ...](var_17,
var_18, var_19, arg_4)
   _ = var_9
   _ = var_7
```

Generated Mojo

Even Runtime boilerplate code is Mojo

```
@export
fn kernel_wrapper_1(arg_27: unsafe_pointer.UnsafePointer[unsafe_pointer.OpaquePointer], arg_28:
unsafe_pointer.OpaquePointer):
   # This is the entry point of the kernel. The first argument is a pointer to a list of inputs and outputs. The
second is a pointer to a runtime context.
   try:
       var var_29 = MOGGPrimitives.mogg_async_unpack[unsafe_pointer.OpaquePointer](arg_27[1])
...
       var var_33 = MOGGPrimitives.mogg_async_unpack[Int](arg_27[5])
...
       var var_36 = MOGGPrimitives.mogg_async_unpack[asyncrt.DeviceContextPtr](arg_27[8])
       alias param_37 = dimlist.DimList.create_unknown[1]()
       alias param_38 = dimlist.DimList(1)
       var var_39 = IndexList[1](1)
       var var_40 = MOGGPrimitives.mogg_tensor_init[DType.float32, 1, False, io_spec.IO.Input, param_37, param_38, 64]
(var_29, var_39)
...

       stub_0(var_40, var_44, var_48, var_52, var_36)

       MOGGPrimitives.mogg_async_ready(arg_27[9])
       MOGGPrimitives.mogg_async_del(arg_27[0])
...
```

Generated Mojo🔥 is human friendly

It can be

introspected

debugged

modified

If you can imagine it in Mojo, it can be done

The road ahead

Today

- Basic fusions
- Partial Mojo codegen (WIP)

Tomorrow
(not in any particular order)

- New fusions
- Megakernels
- Better kernel registration mechanism
- Vertical debugging support (not just Mojo)
- Improved readability of the generated Mojo

We're hiring!

Come build with us at Modular.

If you enjoy programming:

- Compilers
- Runtimes
- Computational kernels
- Serving infrastructures
- All things DL

Modular is the place to be!

modular.com/careers

Thank You

Q&A

Modular