

Modular



Mojo GPU Compilation 🔥

Weiwei Chen
weiwei.chen@modular.com

Abdul Dakkak
adakkak@modular.com

LLVM Developers' Meeting 2025

GPU Programming in Mojo

- Pythonic **systems programming** language
 - Generic programming, type system, and memory safety
 - Blazing fast
 - Best way to extend Python to CPUs and GPUs
 - Bedrock for Modular's MAX inference engine.
- **Unified programming** for CPU + GPU
 - The full power of standard CUDA/ROCm, but without "CUDA"
 - Threads, warps, sync primitives, WMMA instructions
 - Generate executables **without** using vendor toolkits or libraries
 - All GPU kernels for Nvidia, AMD, Apple written in Mojo
- **Library driven** with **simple** compiler support

https://github.com/modular/modular/blob/main/examples/mojo/gpu-functions/vector_addition.mojo

M

GPU kernel

CPU driver code

GPU buffers

GPU device context

compile and launch GPU kernel

device buffer to host

```
from math import ceildiv
from sys import has_accelerator
from gpu import global_idx
from gpu.host import DeviceContext
from layout import Layout, LayoutTensor

alias float_dtype = DType.float32
alias VECTOR_WIDTH = 10
alias BLOCK_SIZE = 5
alias layout = Layout.row_major(VECTOR_WIDTH)

fn vector_addition(
    lhs_tensor: LayoutTensor[float_dtype, layout, MutableAnyOrigin],
    rhs_tensor: LayoutTensor[float_dtype, layout, MutableAnyOrigin],
    out_tensor: LayoutTensor[float_dtype, layout, MutableAnyOrigin],
    size: Int,
):
    """The calculation to perform across the vector on the GPU."""
    var global_tid = global_idx.x
    if global_tid < UInt(size):
        out_tensor[global_tid] = lhs_tensor[global_tid] + rhs_tensor[global_tid]

def main():
    constrained[has_accelerator(), "This example requires a supported GPU"]()

    # Get context for the attached GPU
    var ctx = DeviceContext()

    # Allocate data on the GPU address space
    var lhs_buffer = ctx.enqueue_create_buffer[float_dtype](VECTOR_WIDTH)
    var rhs_buffer = ctx.enqueue_create_buffer[float_dtype](VECTOR_WIDTH)
    var out_buffer = ctx.enqueue_create_buffer[float_dtype](VECTOR_WIDTH)

    # Fill in values across the entire width
    _ = lhs_buffer.enqueue_fill(1.25)
    _ = rhs_buffer.enqueue_fill(2.5)

    # Wrap the device buffers in tensors
    var lhs_tensor = LayoutTensor[float_dtype, layout](lhs_buffer)
    var rhs_tensor = LayoutTensor[float_dtype, layout](rhs_buffer)
    var out_tensor = LayoutTensor[float_dtype, layout](out_buffer)

    # Calculate the number of blocks needed to cover the vector
    var grid_dim = ceildiv(VECTOR_WIDTH, BLOCK_SIZE)

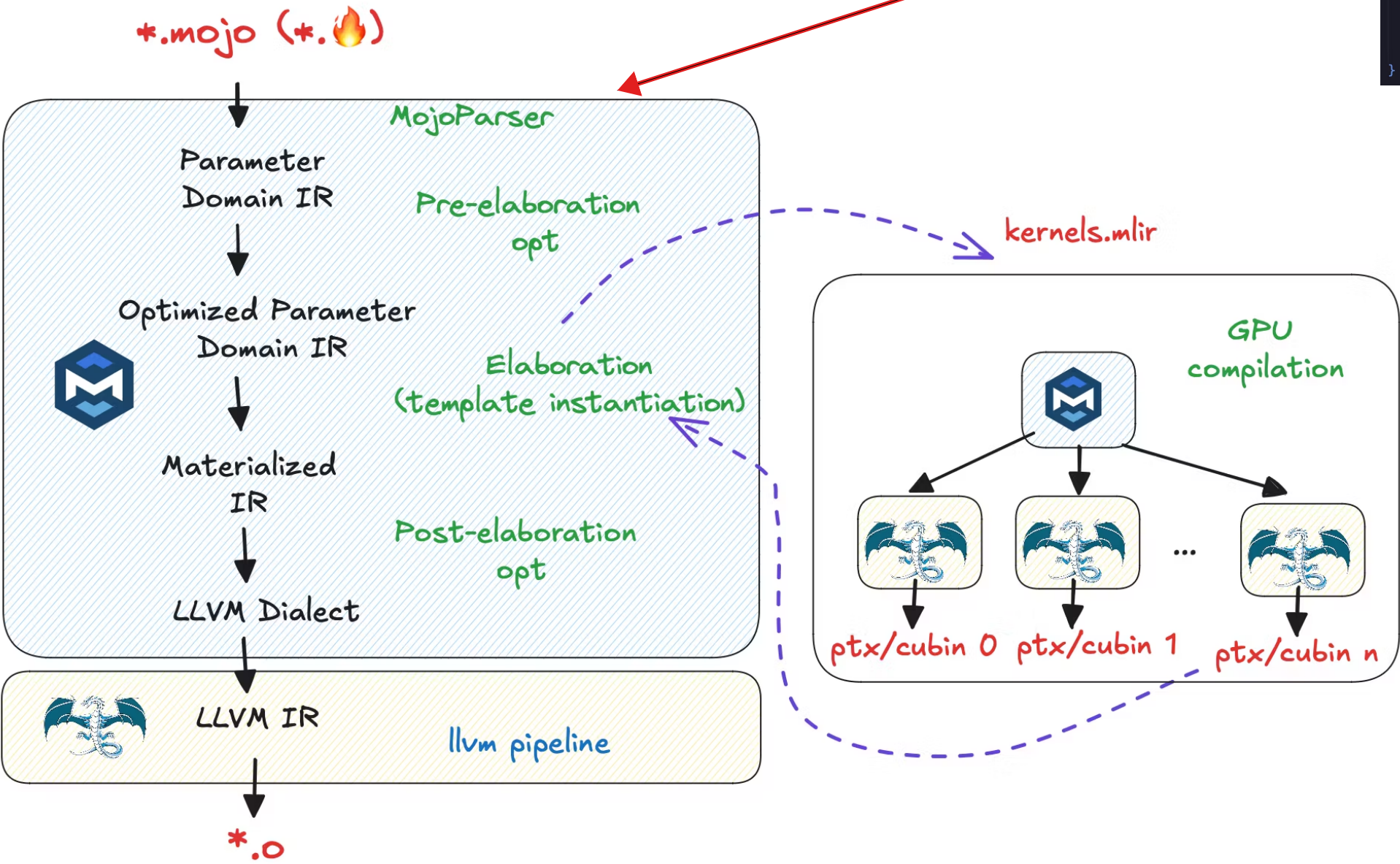
    # Launch the vector_addition function as a GPU kernel
    ctx.enqueue_function_checked(vector_addition, vector_addition)(
        lhs_tensor,
        rhs_tensor,
        out_tensor,
        VECTOR_WIDTH,
        grid_dim=grid_dim,
        block_dim=BLOCK_SIZE,
    )

    # Map to host so that values can be printed from the CPU
    with out_buffer.map_to_host() as host_buffer:
        var host_tensor = LayoutTensor[float_dtype, layout](host_buffer)
        print("Resulting vector:", host_tensor)
```

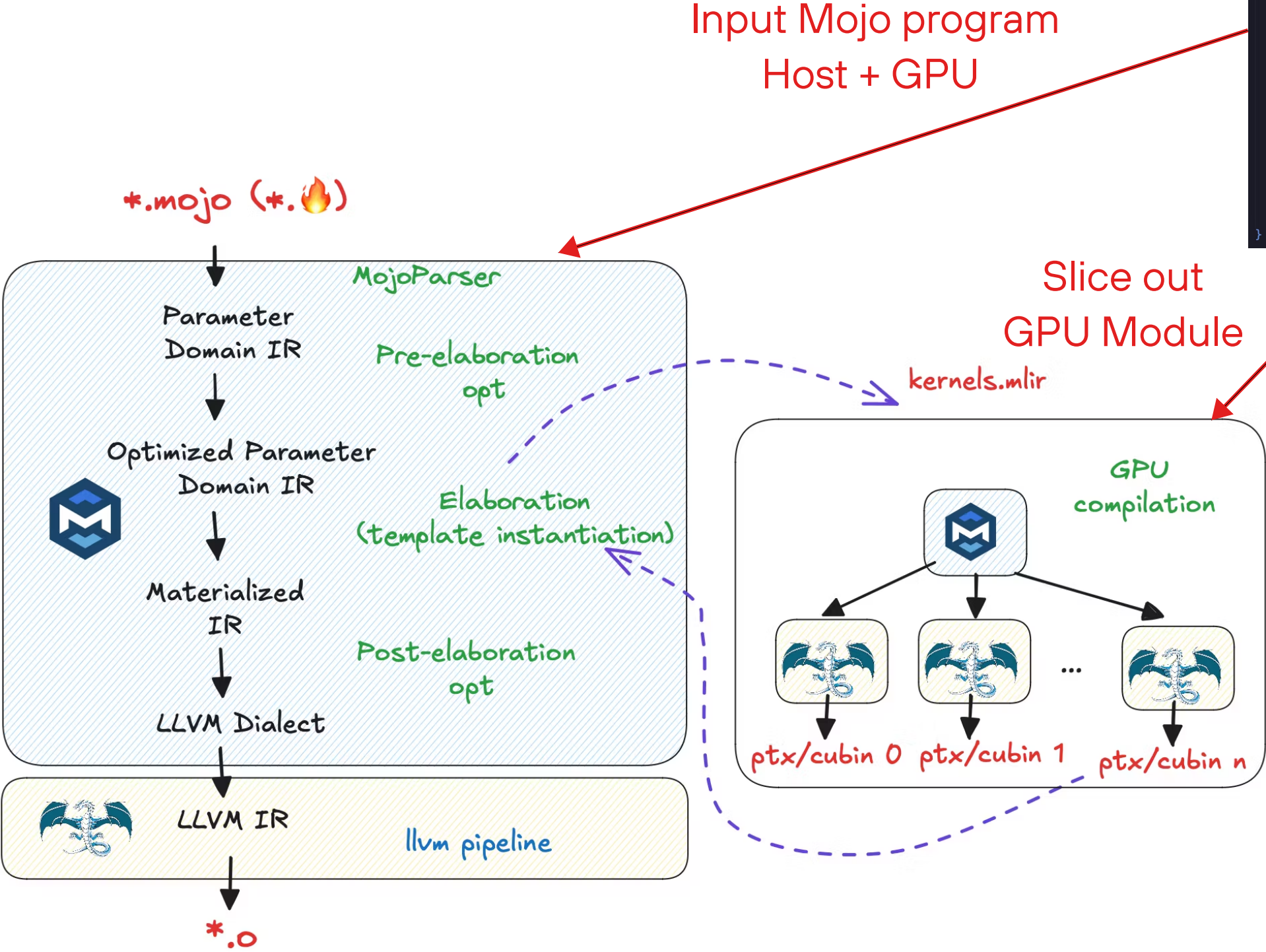
Mojo GPU Compilation Flow

Input Mojo program
Host + GPU

```
module attributes {M.target_info = #M.target<triple = "arm64-apple-darwin23.6.0", arch = "apple-m2", features = "+aes,+bf16,+complexnum,+crc,  
+dotprod,+fp-armv8,+fp16mm,+fpac,+fptoint,+i64mm,+jscnv,+lse,+neon,+pauth,+permon,+ras,+rcpc,+rnm,+sha2,+sha3,+ssbs", data_layout =  
"e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-n32:64-S128-Fn32", relocation_model = "pic", simd_bit_width = 128, index_bit_width =  
64>, kgen.env = #kgen.env<{}>} {  
  kgen.generator @kernel1() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator @kernel2() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator export @entry(%arg0: !kgen.pointer<none>) {  
    kgen.param.declare nvptx: target = <#kgen.target<triple = "nvptx64-nvidia-cuda",  
                                          arch = "sm_80",  
                                          simd_bit_width = 128,  
                                          index_bit_width = 64,  
                                          tune_cpu = "sm_80">>  
    %0:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel1> -> string, index  
    %1:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel2> -> string, index  
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none  
    kgen.call @launch_kernel(%1#0, %1#1) : (!kgen.string, index) -> !kgen.none  
    kgen.return  
  }  
}
```



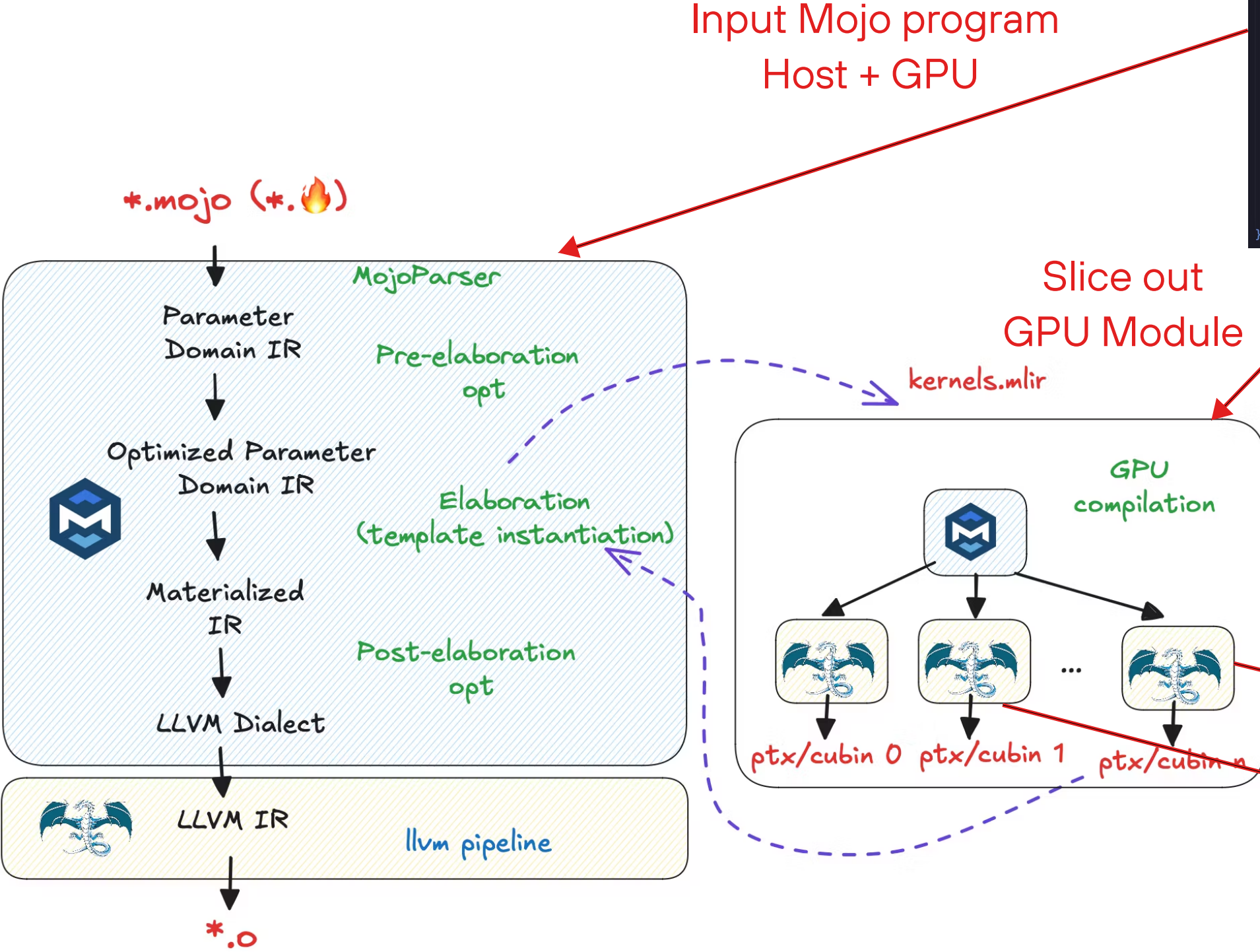
Mojo GPU Compilation Flow



```
module attributes {M.target_info = #M.target<triple = "arm64-apple-darwin23.6.0", arch = "apple-m2", features = "+aes,+bf16,+complexnum,+crc,+dotprod,+fp-armv8,+fp16mat,+fpac,+fullfp16,+l8mm,+lscnv,+lse,+neon,+pauth,+permon,+ras,+rcpc,+rdm,+sha2,+sha3,+ssbs", data_layout = "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-n32:64-S128-Fn32", relocation_model = "pic", simd_bit_width = 128, index_bit_width = 64>, kgen.env = #kgen.env<{}>} {  
  kgen.generator @kernel1() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator @kernel2() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator export @entry(%arg0: !kgen.pointer<none>) {  
    kgen.param.declare nvptx: target = <#kgen.target<triple = "nvptx64-nvidia-cuda", arch = "sm_80", simd_bit_width = 128, index_bit_width = 64, tune_cpu = "sm_80">>  
    %0:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel1> -> string, index  
    %1:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel2> -> string, index  
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none  
    kgen.call @launch_kernel(%1#0, %1#1) : (!kgen.string, index) -> !kgen.none  
    kgen.return  
  }  
}
```

```
module attributes {M.target_info = #M.target<triple = "nvptx64-nvidia-cuda", arch = "sm_80", simd_bit_width = 128, index_bit_width = 64, tune_cpu = "sm_80">, kgen.env = #kgen.env<{}>} {  
  kgen.generator export @kernel1() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 0 : index]} {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator export @kernel2() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 1 : index]} {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
}
```

Mojo GPU Compilation Flow

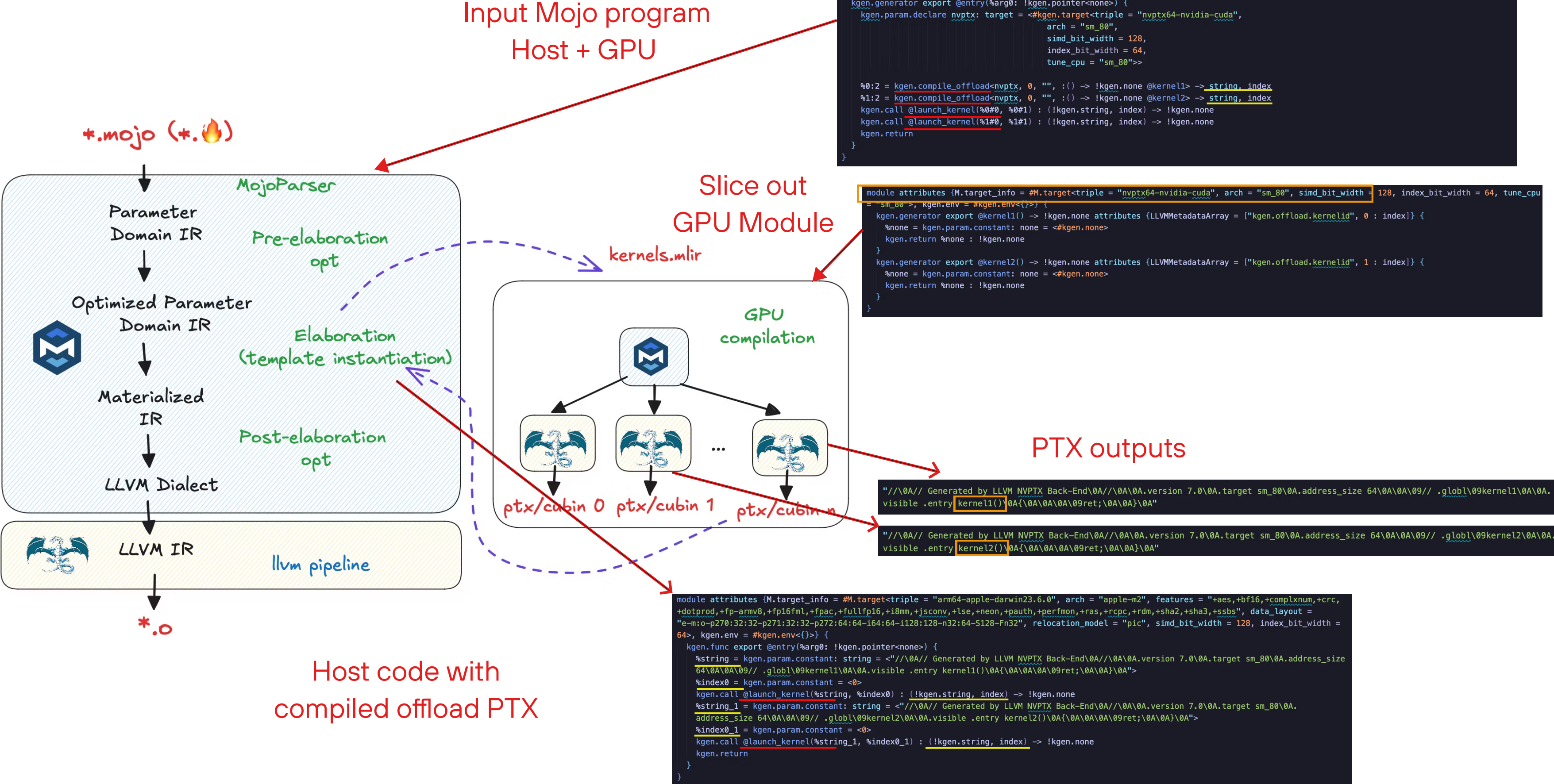


```
module attributes {M.target_info = #M.target<triple = "arm64-apple-darwin23.6.0", arch = "apple-m2", features = "+aes,+bf16,+complexnum,+crc,+dotprod,+fp-armv8,+fp16mat,+fpac,+fullfp16,+lstm,+lse,+neon,+pauth,+permon,+ras,+rcpc,+rdm,+sha2,+sha3,+ssbs", data_layout = "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-n32:64-S128-Fn32", relocation_model = "pic", simd_bit_width = 128, index_bit_width = 64>, kgen.env = #kgen.env<{}>} {  
  kgen.generator @kernel1() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator @kernel2() -> !kgen.none {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator export @entry(%arg0: !kgen.pointer<none>) {  
    kgen.param.declare nvptx: target = <#kgen.target<triple = "nvptx64-nvidia-cuda", arch = "sm_80", simd_bit_width = 128, index_bit_width = 64, tune_cpu = "sm_80">>  
    %0:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel1> -> string, index  
    %1:2 = kgen.compile_offload<nvptx, 0, "", :() -> !kgen.none @kernel2> -> string, index  
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none  
    kgen.call @launch_kernel(%1#0, %1#1) : (!kgen.string, index) -> !kgen.none  
    kgen.return  
  }  
}
```

```
module attributes {M.target_info = #M.target<triple = "nvptx64-nvidia-cuda", arch = "sm_80", simd_bit_width = 128, index_bit_width = 64, tune_cpu = "sm_80">, kgen.env = #kgen.env<{}>} {  
  kgen.generator export @kernel1() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 0 : index]} {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
  kgen.generator export @kernel2() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 1 : index]} {  
    %none = kgen.param.constant: none = <#kgen.none>  
    kgen.return %none : !kgen.none  
  }  
}
```

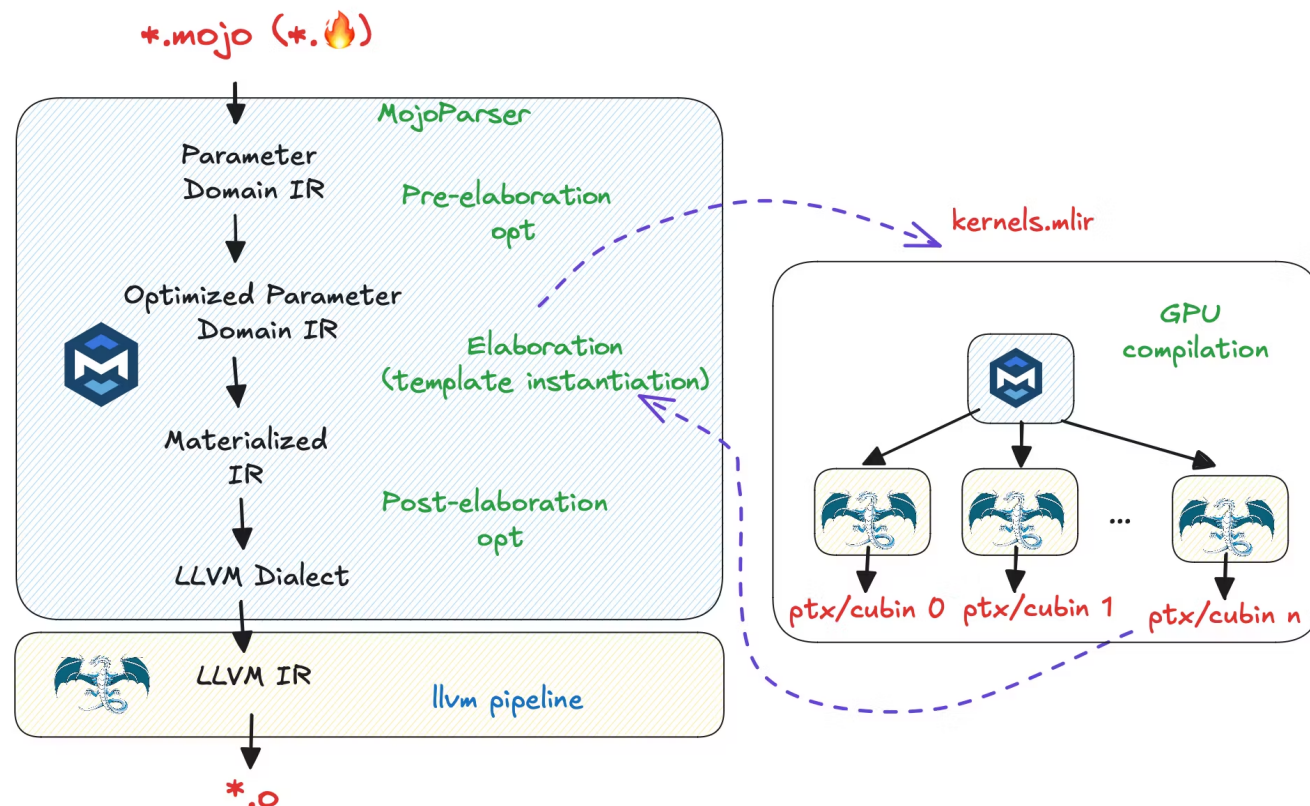
```
"/\0A// Generated by LLVM NVPTX Back-End\0A//\0A\0A.version 7.0\0A.target sm_80\0A.address_size 64\0A\0A\09// .globl\09kernel1\0A\0A.  
visible .entry kernel1()\0A{\0A\0A\0A\09ret;\0A\0A}\0A"  
  
"/\0A// Generated by LLVM NVPTX Back-End\0A//\0A\0A.version 7.0\0A.target sm_80\0A.address_size 64\0A\0A\09// .globl\09kernel2\0A\0A.  
visible .entry kernel2()\0A{\0A\0A\0A\09ret;\0A\0A}\0A"
```

Mojo GPU Compilation Flow



Compiler support for Debugging Kernels 🔍

- Inspecting Mojo GPU kernel in LLVM IR, assembly (PTX), or object file (cubin, hsaco)
- Same GPU compilation flow, just change what the pipeline produces.
- Plug the result as string to host code.



```
//  
// Generated by LLVM NVPTX Back-End  
//  
.version 8.1  
.target sm_80  
.address_size 64  
  
// .globl      compile_offload_hello6A6AoApA  
  
.visible .entry compile_offload_hello6A6AoApA()  
{  
    ret;  
}
```

asm

```
; ModuleID = 'compile_offload.mojo'  
source_filename = "compile_offload.mojo"  
target datalayout =  
  "e-p3:32:32-p4:32:32-p5:32:32-p6:32:32-p7:32:32-i64:64-i128:128-i256:256-v16:16-v32:32-n16:32:64"  
target triple = "nvptx64-nvidia-cuda"  
  
; Function Attrs: norecurse  
define dso_local ptx_kernel void @compile_offload_hello6A6AoApA() #0 {  
    ret void  
}  
  
attributes #0 = { norecurse "target-cpu"="sm_80" "target-features"="+ptx81,sm_80" "tune-cpu"="sm_80" }  
  
!llvm.module.flags = !{!0}  
!0 = !{i32 2, !"Debug Info Version", i32 3}
```

llvm IR

```
; ModuleID = 'compile_offload.mojo'  
source_filename = "compile_offload.mojo"  
target datalayout = "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-n32:64-S128-Fn32"  
target triple = "arm64-apple-darwin23.6.0"  
  
; Function Attrs: mustprogress norecurse nosync nounwind willreturn memory(none)  
define dso_local void @"compile_offload::hello()"() #0 {  
    ret void  
}  
  
attributes #0 = { mustprogress norecurse nosync nounwind willreturn memory(none) "target-cpu"="apple-m2"  
  "target-features"="+aes,+bf16,+complexnum,+crc,+dotprod,+fp-armv8,+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,  
  +neon,+pauth,+perfmon,+ras,+rcpc,+rdm,+sha2,+sha3,+ssbs" }  
  
!llvm.module.flags = !{!0}  
!0 = !{i32 2, !"Debug Info Version", i32 3}
```

opt llvm IR

```
@fieldwise_init  
@register_passable("trivial")  
struct _Info:  
    var kernel: __mlir_type.`!kgen.string`  
    var name: __mlir_type.`!kgen.string`  
    var num_captures: __mlir_type.index  
  
@fieldwise_init  
@register_passable("trivial")  
struct Info:  
    var kernel: StaticString  
    var name: StaticString  
    var num_captures: Int  
  
@always_inline  
fn _compile_info[  
    func_type: AnyTrivialRegType, //,  
    func: func_type,  
    //,  
    emission_kind: StaticString = "asm",  
    target: _TargetType = A100.target(),  
    compile_options: StaticString = CompilationTarget[  
        target  
    ].default_compile_options(),  
]() -> Info:  
    var info = __mlir_op.`kgen.compile_offload`[  
        target_type=target,  
        emission_kind = _get_emission_kind_id[emission_kind]().__mlir_value,  
        emission_option = _get_kgen_string[compile_options](),  
        func=func,  
        _type=_Info,  
    ]()  
  
    return Info(  
        StaticString(info.kernel),  
        StaticString(info.name),  
        Int(mlir.value=info.num_captures),  
    )  
  
fn hello():  
    pass  
  
fn main():  
    # compile kernel to asm.  
    t1 = _compile_info[hello, emission_kind="asm"]()  
    print(t1.kernel)  
  
    # compile kernel to llvm ir.  
    t2 = _compile_info[hello, emission_kind="llvm"]()  
    print(t2.kernel)  
  
    # compile kernel to optimized llvm ir.  
    t3 = _compile_info[hello, emission_kind="llvm-opt"]()  
    print(t3.kernel)
```

Conclusions

- Mojo provides a unified way to write CPU+GPU code => **one MLIR module**
- Library driven GPU feature implementation for different vendors => **simple compiler, no heroic magic**
- MLIR unlocks seamless compiler integration:
 - Native compiler support to know what to slice into GPU mlir modules during elaboration.
 - Add an mlir op `kgen.compile_offload`.
 - Mojo has support as MLIR sugar:
 - No need to change the parser to extend syntax.
 - Ease of writing library code that uses architecture specific dialects (nvvm, rocdl) and low-level intrinsics (llvm).
- Compiling multiple kernels in one MLIR module and split for LLVM pipeline => **fast compilation**
- **Generally applicable** to compile other accelerator offloads built on top of LLVM/MLIR framework.
- Mojo GPU kernels are all open-sourced

<https://github.com/modular/modular/tree/main/max/kernels>



Thank you!
Hope to see you at the poster session!

Acknowledgement:
The Mojo Language Team at Modular

