Modular

Mojo GPU Compilation 🔥

Weiwei Chen
weiwei.chen@modular.com

MLIR Workshop @
LLVM Developers' Meeting 2025

Agenda

01	Mojo in a glance 🔥
02	Writing GPU code in Mojo
03	Mojo GPU Compilation
04	Compiler Support for Kernel Debugging
05	Conclusions

Mojo 🔥 at a glance

Pythonic systems programming language

- CLI toolchain + VS Code support
- Strong Python and C interop
- Extensive generic programming*, type system, and memory safety

Unlocking performance for heterogeneous compute

- Blazing fast
- Unified programming CPU + GPU
- Scalable GPU kernels for Nvidia, AMD, Apple ...

Best way to extend Python to CPUs and GPUs
Backbone for Modular's MAX Inference Engine 👩‍🚀 **

```
def mandelbrot_kernel[
    width: Int  # SIMD Width
](c: ComplexSIMD[float, width]) ->
   SIMD[int, width]:

   """A vectorized implementation of
   the inner mandelbrot computation."""
   z = ComplexSIMD[float, width](0, 0)

   iters = SIMD[DType.index, width](0)
   mask = SIMD[DType.bool, width](True)

   for i in range(MAX_ITERS):
    if not any(mask):
       break

    mask = z.squared_norm() <= 4
    iters = mask.select(iters + 1, iters)
    z = z.squared_add(c)
   return iters
```

Mojo 🔥: A system programming language for heterogenous computing
LLVM Dev Meeting 2023
Mojo : Programming language for all of AI
Mojo: A novel programming language for AI ACAT 202

* 10:30 Tue - Building Modern Language Frontends with MLIR: Lessons from Mojo's Compile-Time Meta-Programming
** 10:30 Wed - Modular MAX's JIT Graph Compiler

GPU Programming in Mojo

GPU kernel in Mojo

Library imports

constants

GPU function

GPU tensor representation

GPU global offset

```mojo
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]
```

CPU host code in Mojo

static assertion

GPU DeviceContext

GPU device buffers

GPU tensors

compile and launch GPU kernel

device buffer to host

```mojo
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)
```

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

GPU Programming in Mojo

- Unified programming for CPU + GPU
- The full power of standard CUDA/ROCm
  - Threads, warps, sync primitives
  - All the WMMA instructions
- "Without CUDA"
  - All GPU kernels written in Mojo.
  - Generate PTX directly without using CUDA toolkits or libraries
  - GPU kernels compiled on demand with small release container size
- GPU kernels for Nvidia, AMD, Apple in one language
- Library driven with "simple" compiler support

```mojo
# ===-----------------------------------------------------------------------===#
# thread_idx
# ===-----------------------------------------------------------------------===#


@register_passable("trivial")
struct _ThreadIdx(Defaultable):
    """ThreadIdx provides static methods for getting the x/y/z coordinates of
    a thread within a block."""

    @always_inline("nodebug")
    fn __init__(out self):
        return

    @always_inline("nodebug")
    @staticmethod
    fn _get_intrinsic_name[dim: StringLiteral]() -> StaticString:
        @parameter
        if is_nvidia_gpu():
            return "llvm.nvvm.read.ptx.sreg.tid." + dim
        elif is_amd_gpu():
            return "llvm.amdgcn.workitem.id." + dim
        elif is_apple_gpu():
            return "llvm.air.thread_position_in_threadgroup." + dim
        else:
            return CompilationTarget.unsupported_target_error[
                StaticString,
                operation="thread_idx field access",
            ]()

    @always_inline("nodebug")
    fn __getattr__[dim: StringLiteral](self) -> UInt:
        """Gets the `x`, `y`, or `z` coordinates of a thread within a block.

        Returns:
            The `x`, `y`, or `z` coordinates of a thread within a block.
        """
        _verify_xyz[dim]()
        alias intrinsic_name = Self._get_intrinsic_name[dim]()
        return UInt(
            llvm_intrinsic[intrinsic_name, UInt32, has_side_effect=False]()
        )


alias thread_idx = _ThreadIdx()
```

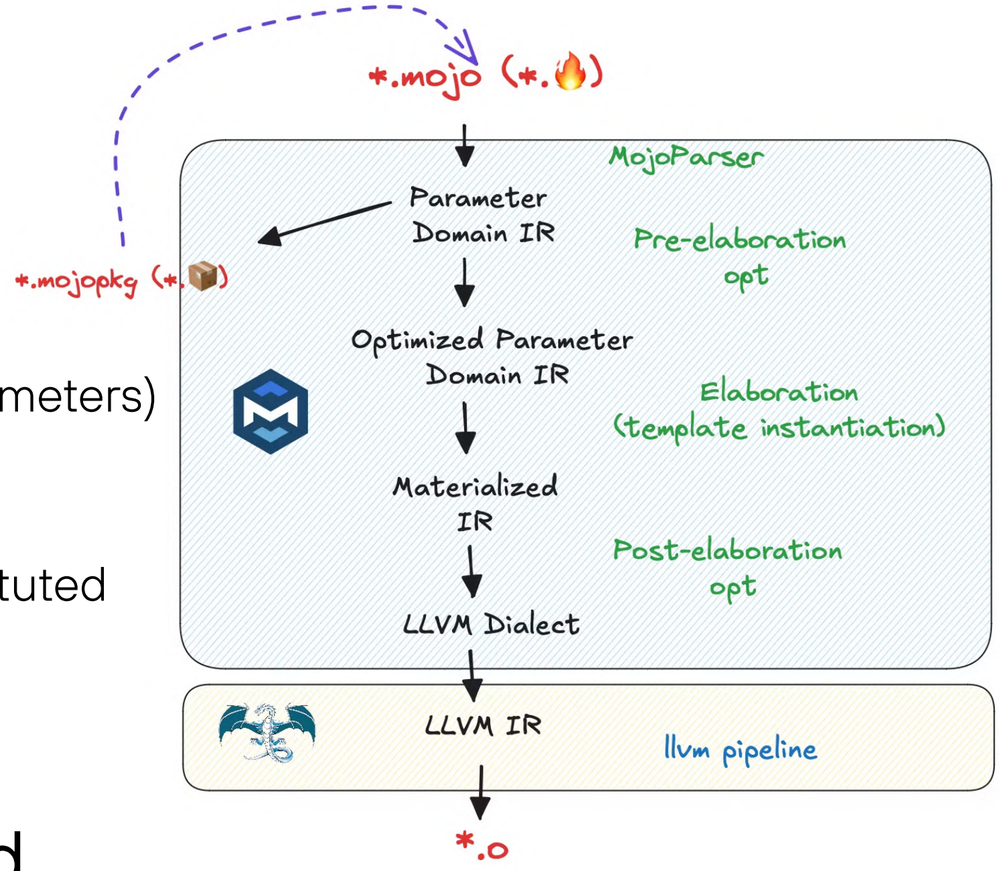
compile-time condition for different HW

llvm intrinsics

https://github.com/modular/modular/blob/main/mojo/stdlib/stdlib/gpu/intrinsics.mojo

Recap: Mojo Compilation Pipeline

- Based on MLIR framework with LLVM as backend.
  - Parser
  - Parameter Domain passes (IR with generics/parameters)
  - Elaboration
  - Materialized IR optimizations (Parameters substituted with concrete values)
- Parallelized LLVM backend for Fast compilation time + Performant generated code.

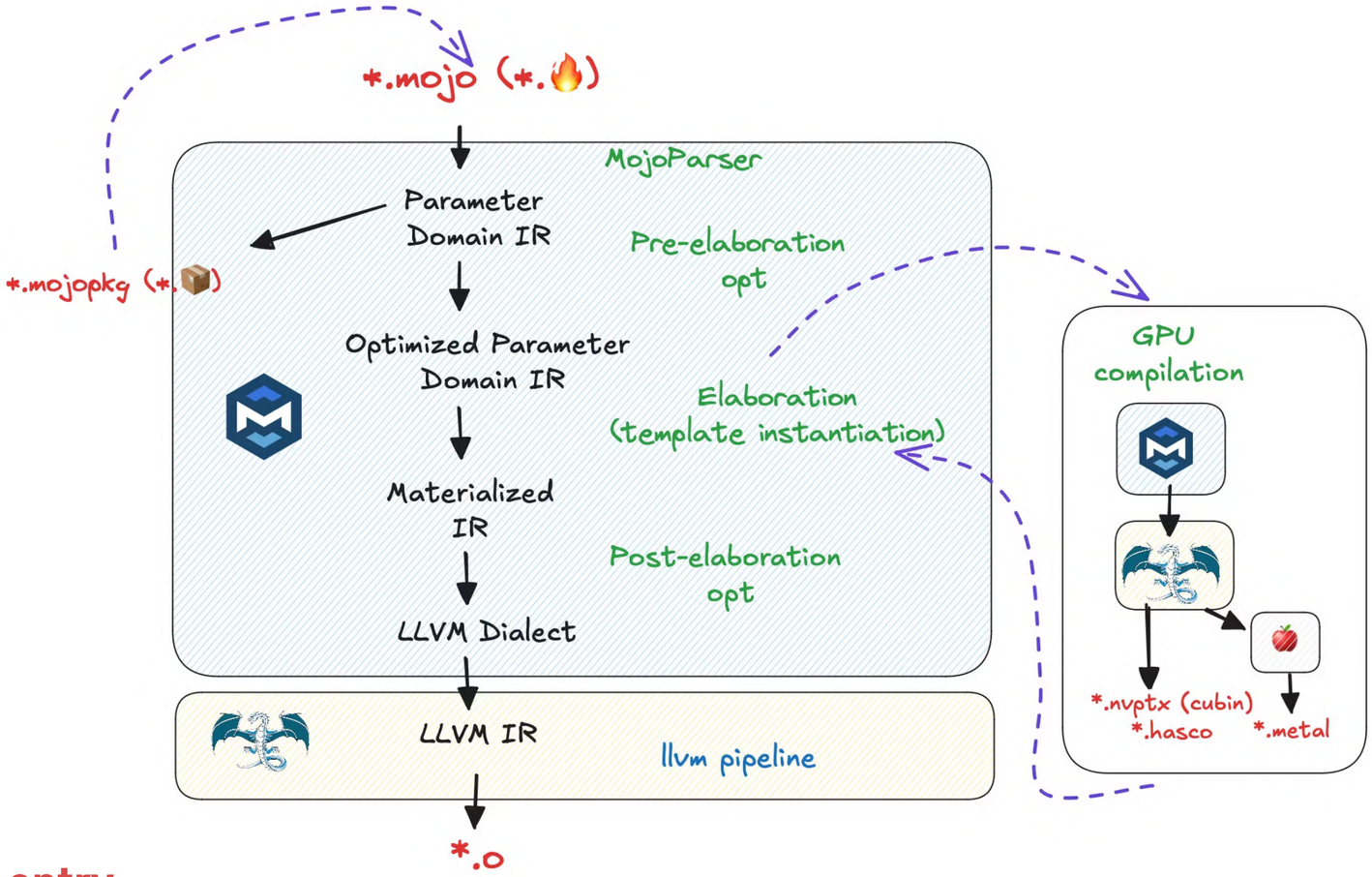


What we've learned from building Mojo's optimization pipeline *@ LLVM Dev 2024*
Parallelizing the LLVM Pipeline with MCLink *@ EuroLLVM 2025*

Mojo GPU Compilation Flow

```mojo
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)
```

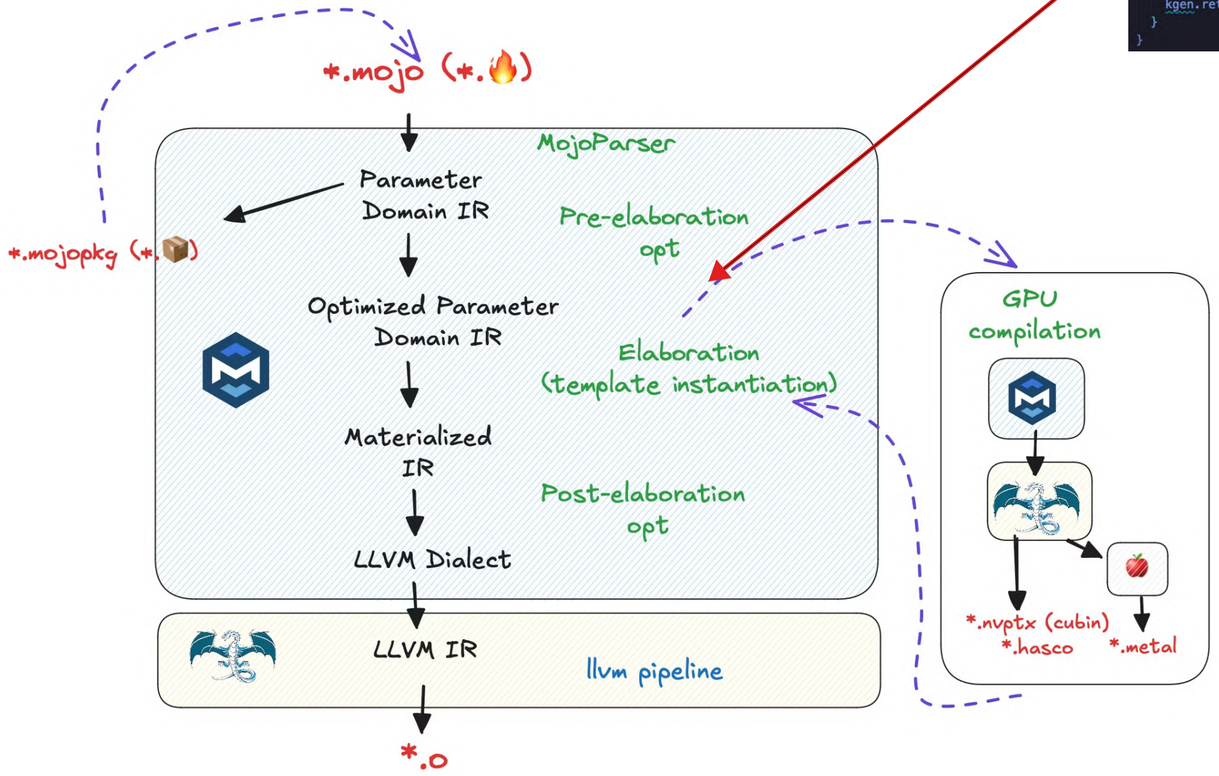
GPU entry function as a parameter



Mojo GPU Compilation Flow

Input Mojo program
Host + GPU

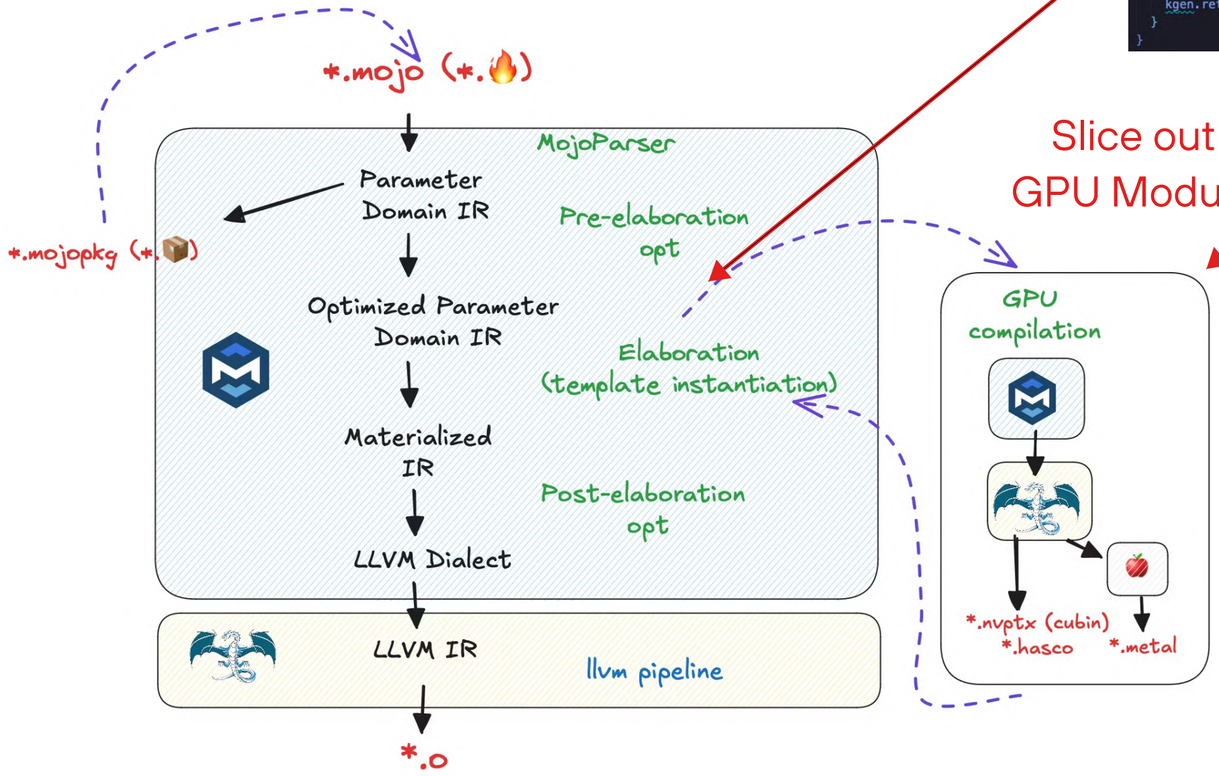
```
module attributes {M.target_info = #M.target<triple = "arm64-apple-darwin23.6.0", arch = "apple-m2", features = "+aes,+bf16,+complxnum,+crc,+dotprod,+fp-armv8,
+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,+neon,+pauth,+perfmon,+ras,+rcpc,+rdm,+sha2,+sha3,+ssbs", data_layout =
"e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-n32:64-S128-Fn32", relocation_model = "pic", simd_bit_width = 128, index_bit_width = 64>, kgen.env = #kgen.
env<{}>} {
  kgen.generator @hello() -> !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 @hello> -> !kgen.string, index
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none
    kgen.return
  }
}
```



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,+complxnum,+crc,+dotprod,+fp-armv8,
+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,+neon,+pauth,+perfmon,+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 @hello() -> !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 @hello> -> !kgen.string, index
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none
    kgen.return
  }
}
```



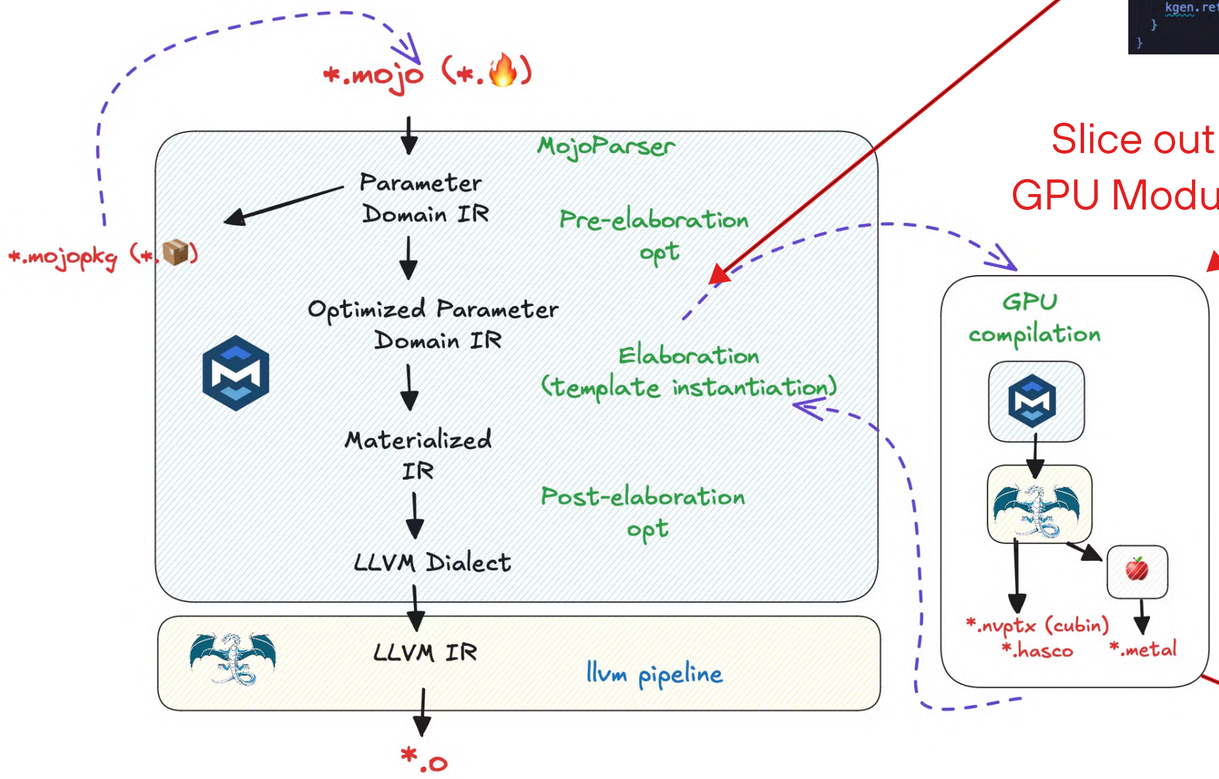
Slice out
GPU Module

```
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 @hello() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 0 :
  index]} {
    %none = kgen.param.constant: none = <#kgen.none>
    kgen.return %none : !kgen.none
  }
}
```

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,+complxnum,+crc,+dotprod,+fp-armv8,
+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,+neon,+pauth,+perfmon,+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 @hello() -> !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 @hello> -> !kgen.string, index
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none
    kgen.return
  }
}
```



Slice out
GPU Module

```
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 @hello() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 0 :
  index]} {
    %none = kgen.param.constant: none = <#kgen.none>
    kgen.return %none : !kgen.none
  }
}
```

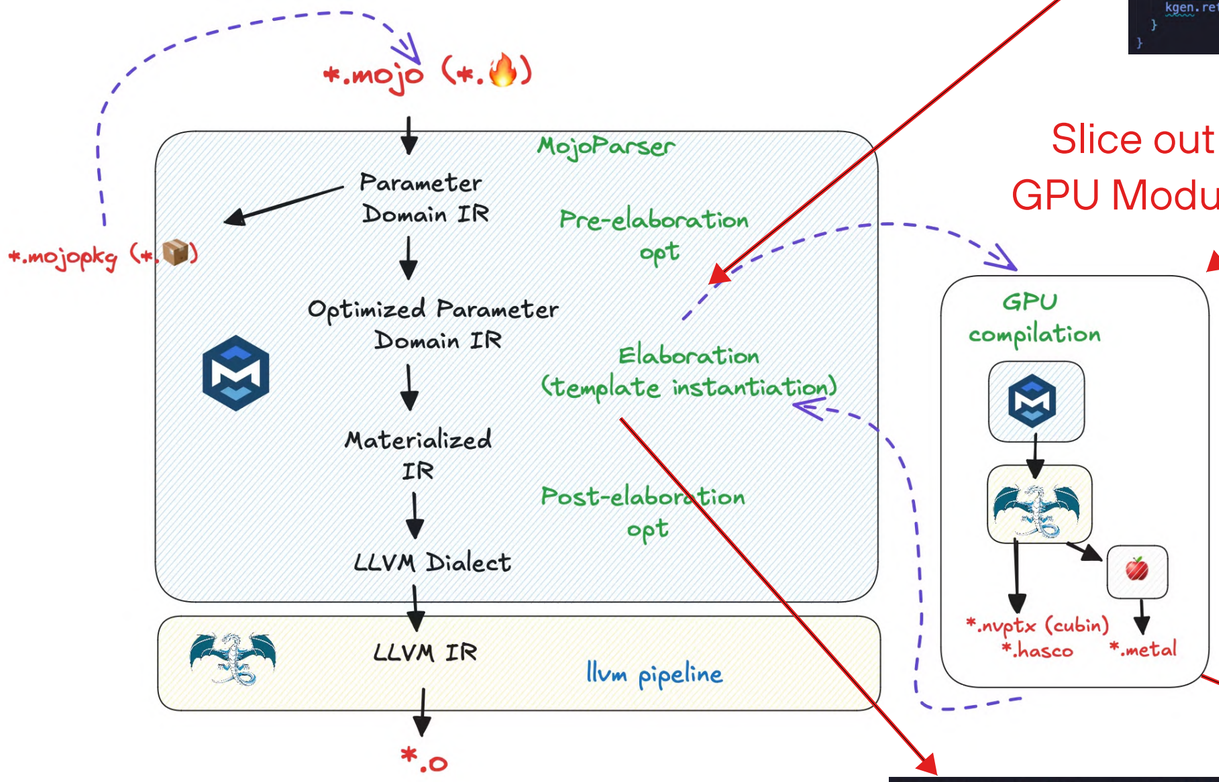

PTX output

```
"//\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\09hello\0A\0A.visible .entry hello()\0A{\0A\0A\0A\09ret;\0A\0A}\0A"
```

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,+complxnum,+crc,+dotprod,+fp-armv8,
+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,+neon,+pauth,+perfmon,+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 @hello() -> !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 @hello> -> !kgen.string, index
    kgen.call @launch_kernel(%0#0, %0#1) : (!kgen.string, index) -> !kgen.none
    kgen.return
  }
}
```



Slice out
GPU Module

```
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 @hello() -> !kgen.none attributes {LLVMMetadataArray = ["kgen.offload.kernelid", 0 :
  index]} {
    %none = kgen.param.constant: none = <#kgen.none>
    kgen.return %none : !kgen.none
  }
}
```


PTX output

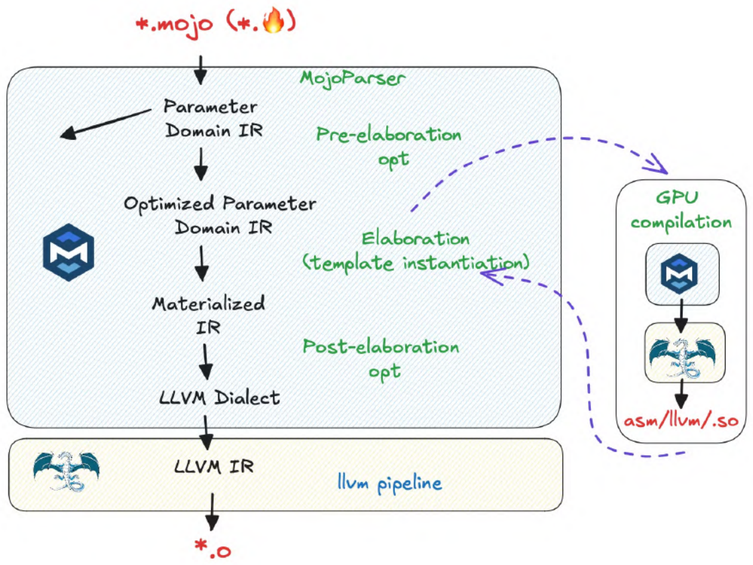
```
"//\0A// Generated by LLVM NVPTX Back-End\0A//\0A\0A.version 7.0\0A.target sm_80\0A.address_size 64\0A\0A\09// .
globl\09hello\0A\0A.visible .entry hello()\0A{\0A\0A\0A\09ret;\0A\0A}\0A"
```

Host code with
compiled offload PTX

```
module attributes {M.target_info = #M.target<triple = "arm64-apple-darwin23.6.0", arch = "apple-m2", features = "+aes,+bf16,+complxnum,+crc,+dotprod,+fp-armv8,
+fp16fml,+fpac,+fullfp16,+i8mm,+jsconv,+lse,+neon,+pauth,+perfmon,+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.func export @entry(%arg0: !kgen.pointer<none>) {
    %string = kgen.param.constant: string = <"//\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\09hello\0A\0A.visible .entry hello()\0A{\0A\0A\0A\09ret;\0A\0A}\0A">
    %index0 = kgen.param.constant = <0>
    kgen.call @launch_kernel(%string, %index0) : (!kgen.string, index) -> !kgen.none
    kgen.return
  }
}
```

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 nofree norecurse nosync nounwind willreturn memory(none)
define dso_local void @"compile_offload::hello()"() #0 {
  ret void
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind willreturn memory(none) "target-cpu"="apple-m2"
"target-features"="+aes,+bf16,+complxnum,+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}
```

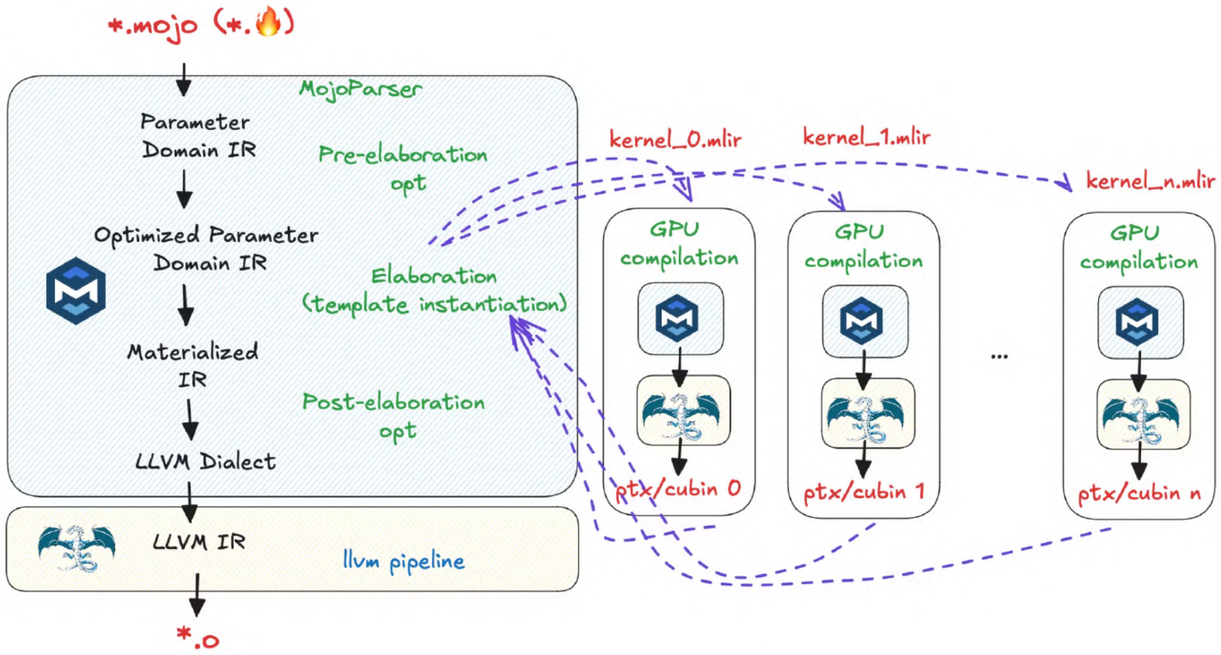
opt llvm IR

```
@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)
```

https://github.com/modular/modular/blob/main/mojo/stdlib/test/compile/test_compile.mojo

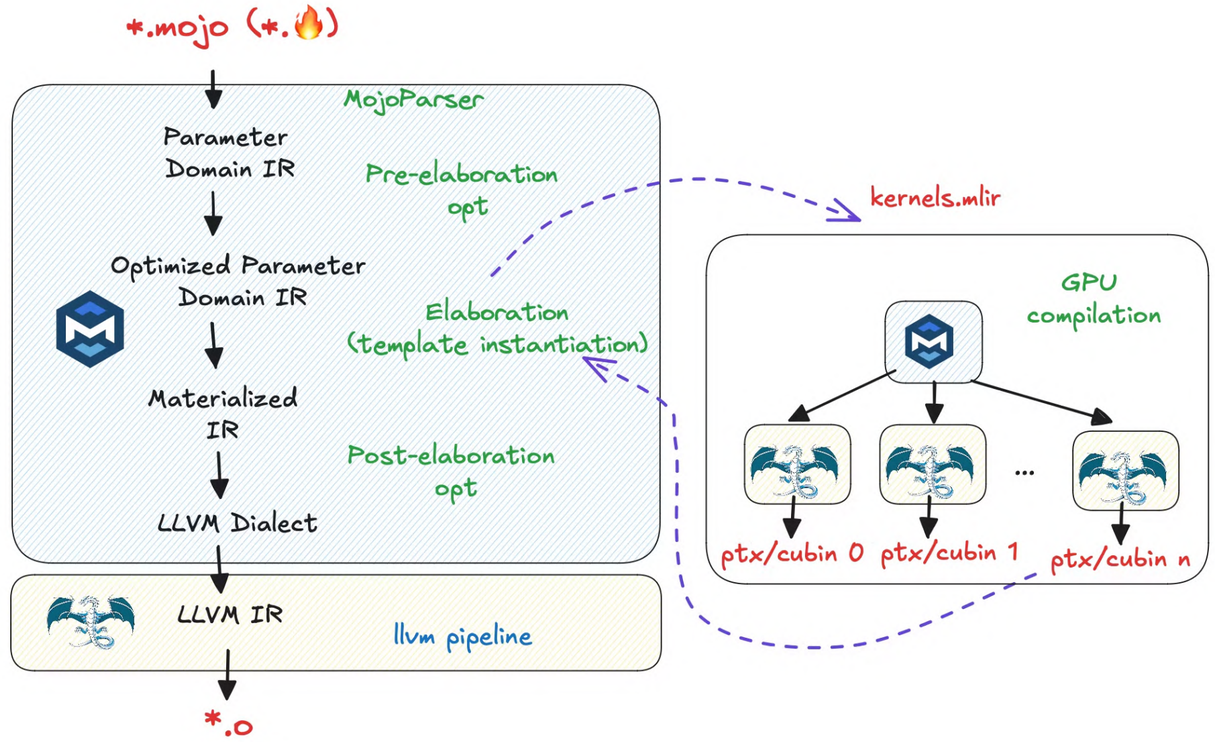
Per-kernel GPU compilation 🌰🌰🌰

- Multiple GPU kernels in one Mojo program or an MAX Model.
  - Slice each kernel out to run full GPU pipeline.
  - Ease to debug each kernel for lowering from input MLIR to LLVM. ✅
  - Efficient to cache compiled kernels if they will be used in another program or model. ✅
  - Logically easy to maintain.



All-kernel GPU compilation 🌰

- Multiple GPU kernels in one Mojo program or an MAX Model.
  - Slice out one GPU model with all the GPU kernels.
  - Only run MLIR pipeline once instead of per kernel. ✅
  - Split each kernel into separate LLVM modules to run backend pipeline in parallel. * ✅
  - Can cache LLVM compilation for kernels if they will be used in another program or model.
  - Faster compilation time.



* ***Parallelizing the LLVM Pipeline with MCLink*** *@ EuroLLVM 2025*

Compilation Time Comparison

/usr/bin/time mojo build Kernels/test/gpu/linalg/test_matmul.mojo

g5.8xlarge

	All-kernel GPU compilation flow					Per-kernel GPU compilation flow						
opt, dbg-level	user (s)	system (s)	elapsed	CPU	maxresident(k)	user (s)	system (s)	elapsed	CPU	maxresident(k)	user+sys old/new x	maxres new/old x
O3, none	236.89	9.40	0:51.07	482%	3138480	499.05	18.17	0:56.40	917%	4540640	**2.10**	**0.691197716621445**
O3, line-tables	292.20	35.73	1:08.71	477%	36492336	590.43	22.34	1:05.53	934%	5295764	**1.87**	6.89085389756794
O2, none	259.18	10.59	0:52.61	512%	3604328	585.77	21.04	1:02.15	976%	5484716	**2.25**	**0.657158547498175**
O2, line-tables	319.19	37.70	1:11.42	499%	38531356	679.10	25.24	1:11.65	983%	6226044	**1.97**	6.18873814576318

/usr/bin/time mojo build Kernels/test/gpu/linalg/test_linalg_matmul_gpu.mojo

g5.8xlarge

	All-kernel GPU compilation flow					Per-kernel GPU compilation flow						
opt, dbg-level	user (s)	system (s)	elapsed	CPU	maxresident(k)	user (s)	system (s)	elapsed	CPU	maxresident(k)	usr+sys old/new x	maxres new/old x
O3, none	49.44	2.03	0:20.95	245%	833268	60.36	3.80	0:20.67	310%	1775324	**1.25**	**0.469361085638452**
O3, line-tables	57.53	3.00	0:23.99	252%	1450892	70.53	4.16	0:23.45	318%	2045704	**1.23**	**0.709238482204659**
O2, none	52.37	2.29	0:21.30	256%	919580	66.36	3.94	0:21.27	330%	1998244	**1.29**	**0.46019405037623**
O2, line-tables	59.88	3.32	0:23.95	263%	1546068	75.69	4.75	0:23.45	342%	2238268	**1.27**	**0.690743020942979**

MODULAR_MAX_TEMPS_DIR=/tmp/llama3_artifacts flowmeter -n 1 utils/benchmarking/flowmeter/pipelines/max-llama3_1-python-gpu.yaml

a100

	input_toks	output_toks	startup_ms	TTFT_ms	CE_tps	Time per Output Token	TG_tps	Total Latency	Total Throughput
All-kernel	482	128	61263.26	2738.74	175.99	13.06	76.57	4397.29	0.23
Per-kernel	482	128	227997.04	3376.99	142.73	13.03	76.74	5031.90	0.20
Per-kernel/ All-kernel(x)			**3.72**	**1.23**	**0.81**	**1.00**	**1.00**	**1.14**	**0.87**

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!

Acknowledgement:
The Mojo Language Team at Modular