Modular

What we've learned from building Mojo🔥's optimization pipeline

Weiwei Chen
weiwei.chen@modular.com

Jeff Niu
jeff@modular.com

LLVM Dev 2024

Agenda

01 Mojo🔥 at a glance

02 Mojo Compilation Pipeline

03 MLIR pipeline and what we've learned

04 LLVM pipeline and what we've learned

05 Conclusions

Mojo 🔥 at a glance

- Pythonic system programming language
- Extensive generic programming and type system
  - Parameters
  - Trait Inheritance
  - Generic Functions and Collections
- Safe and efficient memory model
  - Lifetimes and provenance checking
  - Linear Types *
- C-level control and performance
  - @unroll, @always_inline
  - Mojo 🔥 - A journey to 68,000x speedup over Python
  - First-class library support for extending compiler **
- Heterogenous programming
  - Unified programing for CPU+GPU+... ***
  - Asynchronous Programming ****

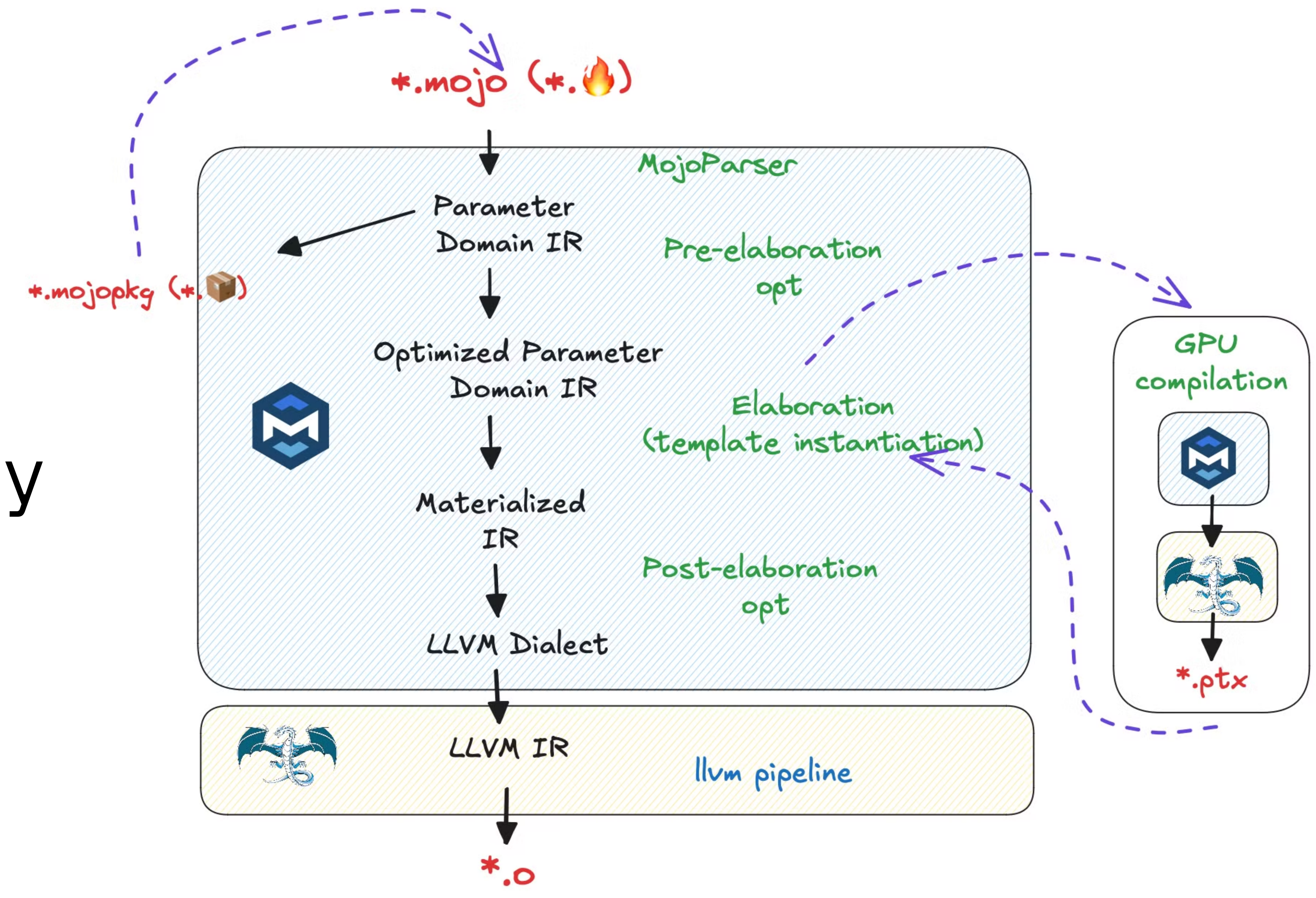
```
# Leverage compile-time meta-programming to write hardware-agnostic
# algorithms and reduce boilerplate.
def exp[dt: DType, elts: Int](x: SIMD[dt, elts]) -> SIMD[dt, elts]:
    x = clamp(x, -88.3762626647, 88.37626266)
    k = floor(x * INV_LN2 + 0.5)
    r = k * NEG_LN2 + x
    return ldexp(_exp_taylor(r), k)

# Take control of storage by inline-allocating values into structures.
struct MyPair(Stringable): # struct with trait(s)
    var first: Int32
    var second: Float32

    def __init__(inout self, first: Int32, second: Float32):
        self.first = first
        self.second = second

    fn __str__(self) -> String: # __str__ for Stringable
        return "[" + self.first.__str__() + "," + self.second.__str__() + "]"


# Take advantage of memory safety without the rough edges.
def reorder_and_process(owned x: HugeArray):
    sort(x)        # Update in place

    give_away(x^) # Transfer ownership

    print(x[0])   # Error: 'x' moved away!
```

** 4:45 tomorrow - Implementing Linear / Non-destructible Types in Vale and Mojo*
*** 4:45 today - Unlocking High Performance in Mojo through User-Defined Dialects*
**** 2:45 tomorrow - Simplifying GPU Programming with Parametric Tile-Level Tensors In Mojo*
***** 2:45 today - Efficient Coroutine Implementation in MLIR*

Mojo 🔥: A system programming language for heterogenous computing
LLVM Dev 2023
Mojo : Programming language for all of AI modular.com/mojo
Mojo: A novel programming language for AI ACAT 202

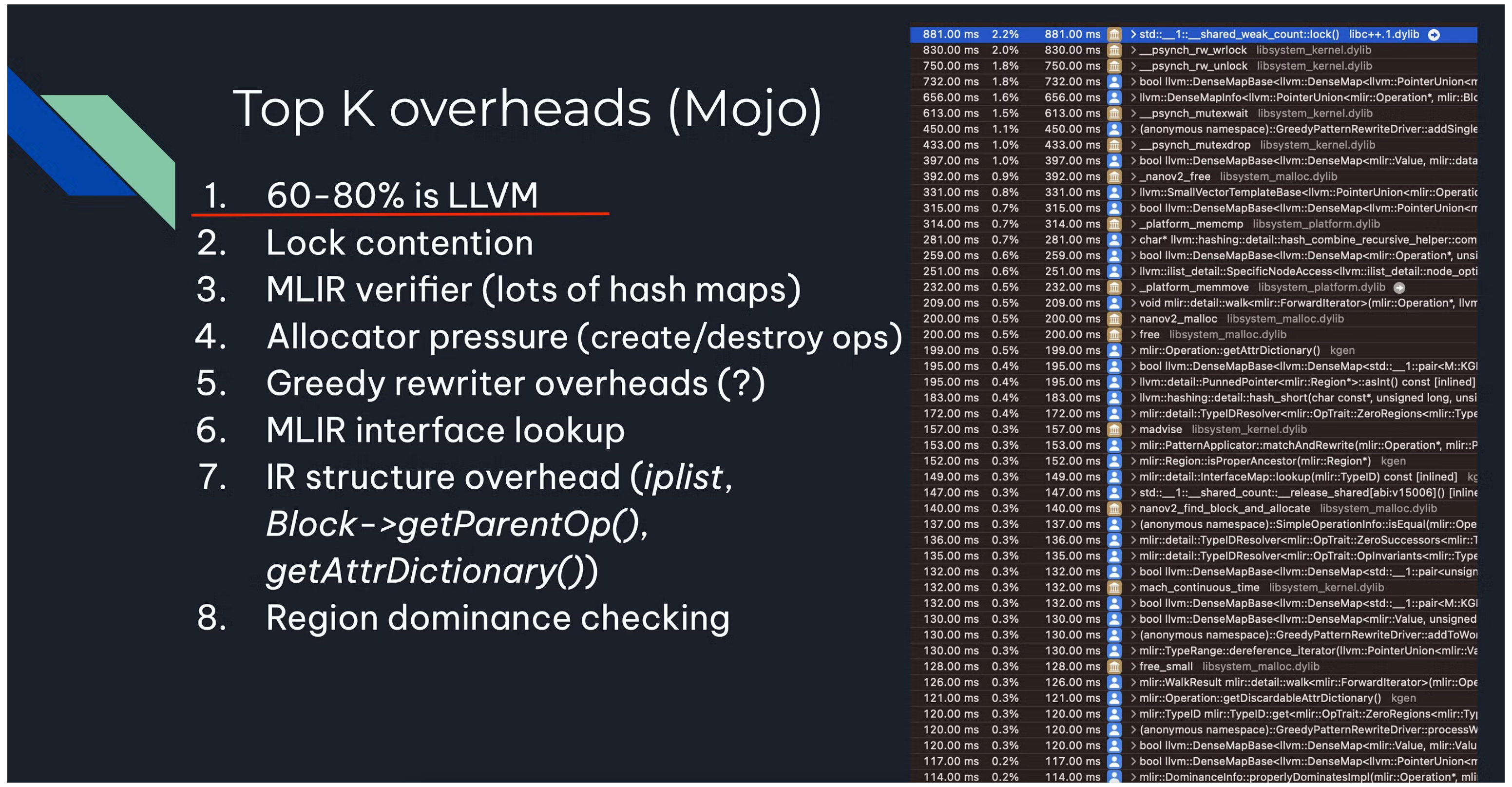
Mojo Compilation Pipeline

- Based on MLIR framework with LLVM as backend.
  - Mojo parser != Clang
  - New MLIR passes!
  - Using LLVM unconventionally
- Library driven compilation for heterogeneous platforms.
- 🎯 **Fast** compilation time + **Performant** generated code.

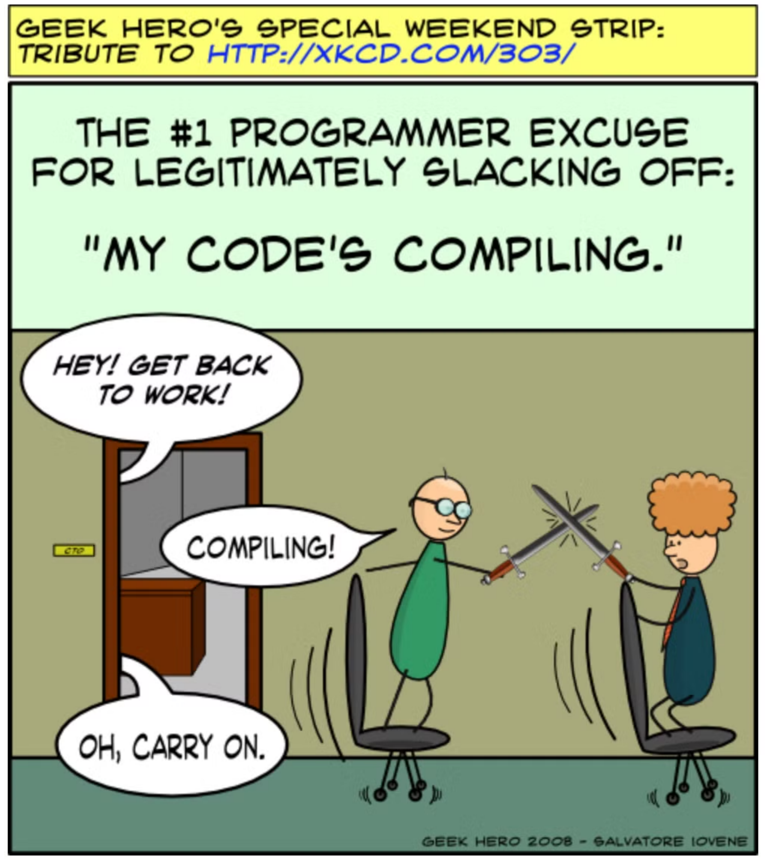


What we've found with Mojo's compilation time

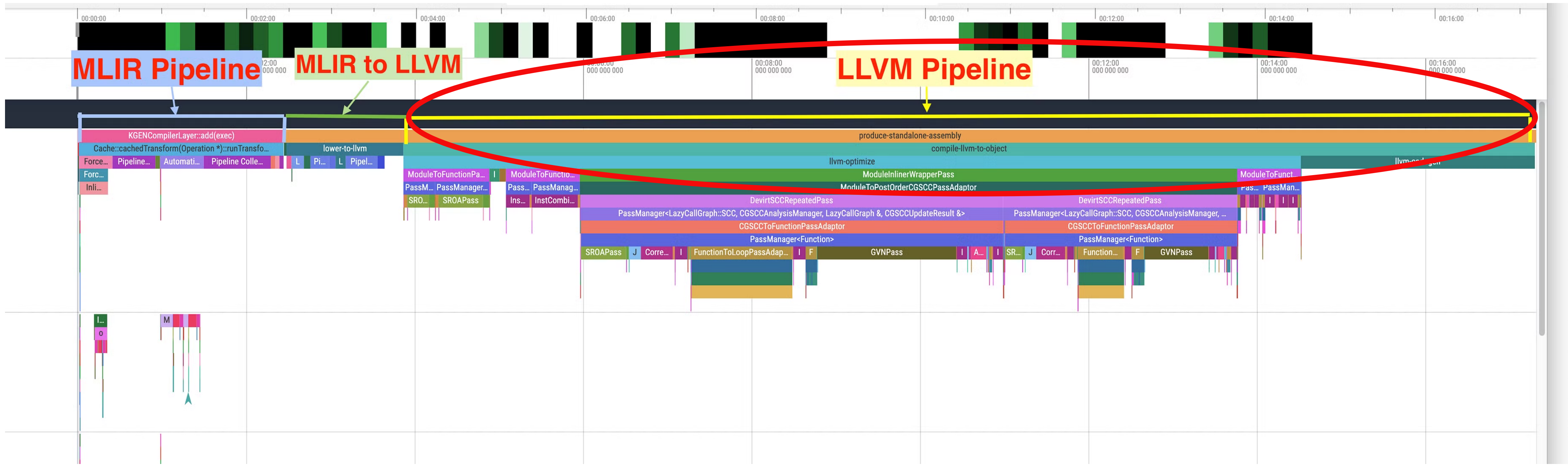
- LLVM Pipeline takes 60~80% Mojo compilation time. 😵‍💫
- -O0 compilation time is an order of magnitude slower than -O3 (for GPU). 🤯



How Slow is MLIR? by Mehdi Amini, Jeff Niu, *EuroLLVM 2024*

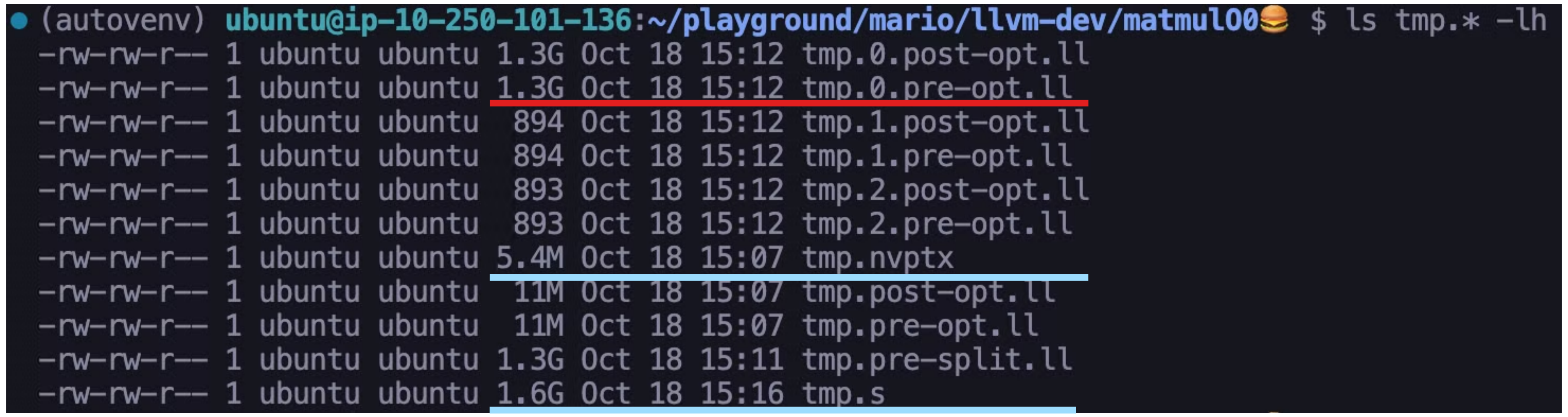


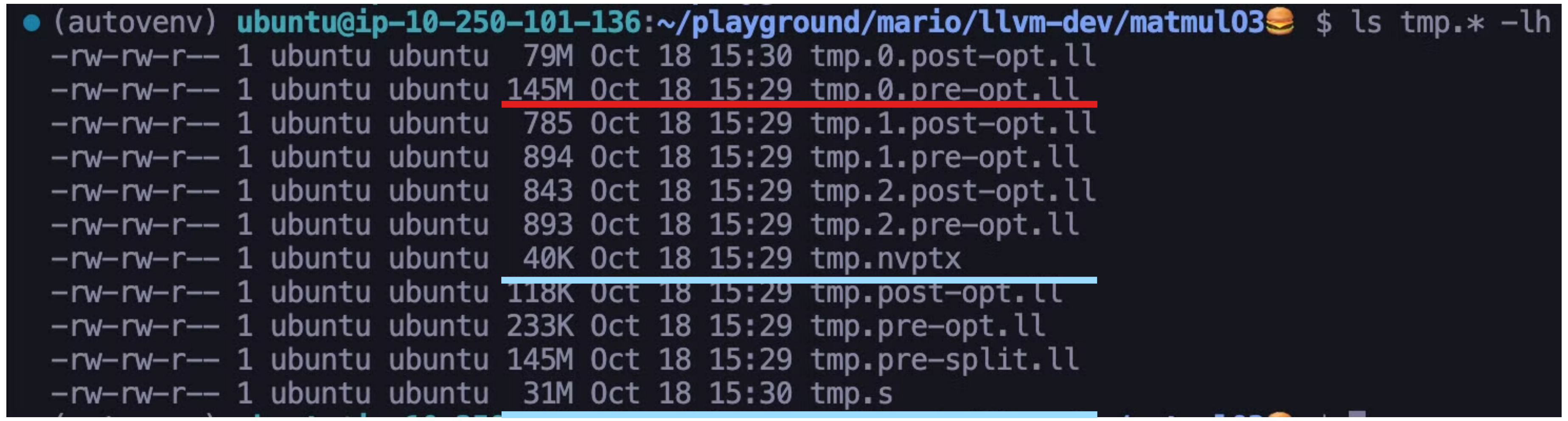
PC: Geek Hero Comic



What we've learned

- Mojo has extensive generic programming support which leads to large IR size.
- -O0 MLIR IR size is significantly larger (5~10x) than -O3
  - Pressure on compile time interpreter
  - Pressure on MLIR passes.
  - Slows down the LLVM pipeline.





Mojo MLIR Pipeline

⚡ Compilation Time 📦 IR Size 🚁 Performant Code

- 🚁⚡📦 Replace (some) LLVM optimization passes and grinding down IR size.
- Optimization passes:
  - 🚁📦 Mem2Reg, SROA, SCCP, LoopUnrolling, StackReuse, SimplifyCF
  - ⚡🚁 Functionality Lowering passes for closures, async coroutines*, etc.
  - 📦🚁 Canonicalizer, CSE, EliminateDeadSymbols, DeadArgumentElimination, etc.

** 2:45 today - Efficient Coroutine Implementation in MLIR*

Higher-level IR representations in MLIR

- ⚡📦🚁 Higher-level IR representing multiple levels of abstractions (e.g: async coroutine, stack allocation, variadic packs, etc) helps to simplify optimization pass implementation (Mem2Reg, SROA, etc).
- Region-based structured control flow
  - early exits: break, continue that exits in the middle of basic blocks.
  - High-level control flow representation matches well with program logic.
  - ⚡📦🚁Guarantees best case scenario for many dataflow analysis and passes (SCCP, Mem2Reg) *.

```
func.func @foobar() {
  rcf.loop {
    %0 = call @rand_bool() : () → i1
    rcf.if %0 {
      rcf.break
    }
    call @do_something() : () → ()
    rcf.continue
  }
  return
}
```

* [Efficient Data-Flow Analysis on Region-based Control Flow in MLIR] EuroLLVM 2024

⚡Compilation Time 📦 IR Size 🚁 Performant Code

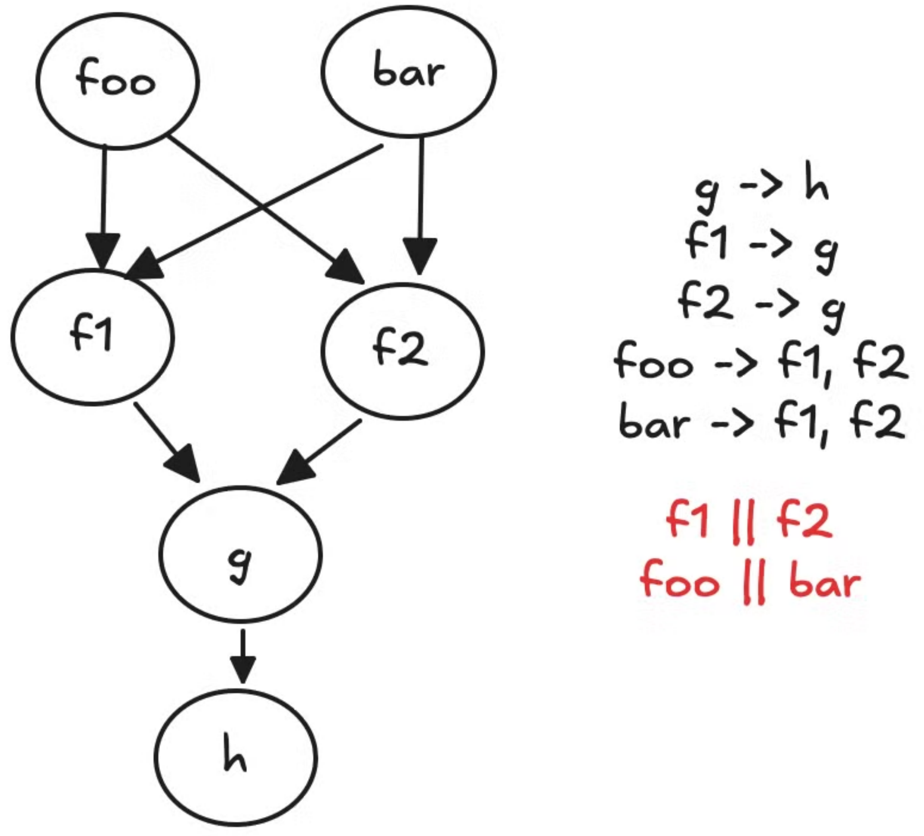
Mojo MLIR Pipeline Parallelization

⚡Compilation Time 📦 IR Size 🚁 Performant Code

- ⚡MLIR framework parallel support for FuncOp passes.

```
pm.addNestedPass<FuncOp>(pass: createSROA());
pm.addNestedPass<FuncOp>(pass: createMem2Reg());
pm.addNestedPass<FuncOp>(pass: createSCCP());
pm.addNestedPass<FuncOp>(pass: createCanonicalizer());
pm.addNestedPass<FuncOp>(pass: mlir::createCSEPass());
```

- ⚡Implement IPO passes with intra-pass parallelization:
  - 🚁⚡Parallel inliner with CallGaph dependency + inner Function level pipeline.
  - ⚡📦 Parallel elaboration: bytecode interpreter, eliminate un-used parameters*



* **MLIR Interpreter for a stack-based programming language** by Jeff Niu, *MLIR Workshop @ LLVM Dev 2023*

Mojo LLVM Pipeline

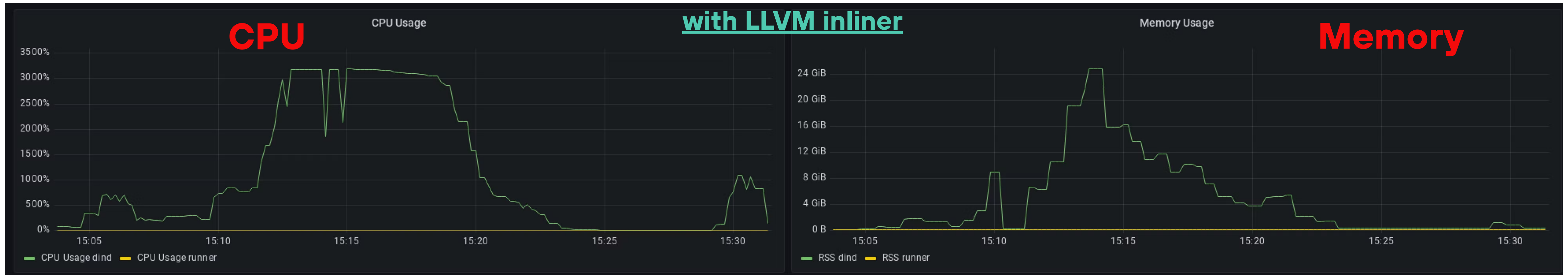
- LLVM is 🙂:
  - GVN, Load/Store Optimization, LSR, scalar optimization, etc.
  - Target-specific code generation.
- LLVM is 😞:
  - Weak and unpredictable loop optimizations.
  - Not parallelized to leverage emerging/modern machines.
- Simplify LLVM Pipeline to do less work:
  - Disable vectorizer, loop-unroller (move to mojo library).
  - Disable Coro-related passes (move to MLIR).
  - Disable all IPO passes (performance?).
- Make LLVM pipeline function pass only and parallelize. (?)

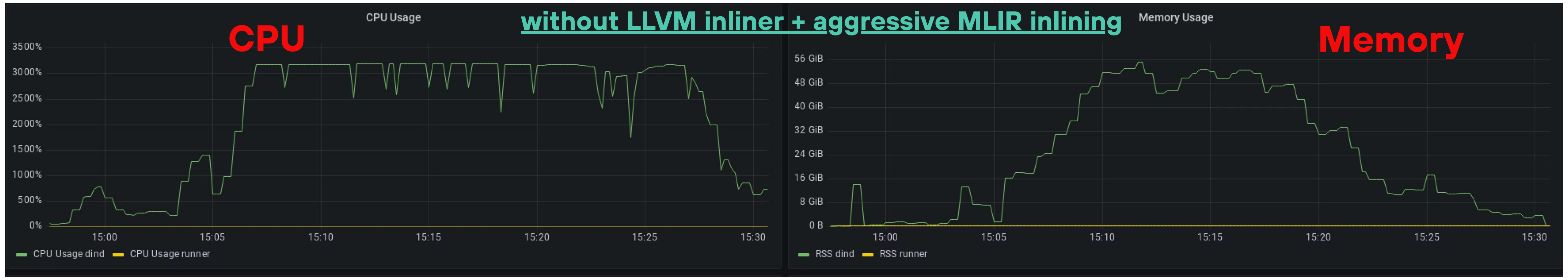
What we've learned

- Inlining is critical for generated code performance:
  - ~2x slow down without LLVM inliner.
  - Recovers performance with aggressive inlining
    - Pays cost of memory for code size.
    - Doesn't necessarily speeds up compilation time.
  - We still need LLVM inliner (till we build an equally sophisticated one in MLIR).

Model: `resnet50-v1.5`, instance type: `c5.4xlarge`, framework: `modular-tensorflow`

Metric	This PR	Modular Main
mean	23303384.0	11091440 (0.48x) ❌
p50	22196377.0	10509161 (0.47x) ❌
p90	26161145.0	14467093 (0.55x) ❌
p95	33135690.0	14948850 (0.45x) ❌
p97	33960179.0	15039500 (0.44x) ❌
p99	34451438.0	15356818 (0.45x) ❌
p999	35475040.0	15954304 (0.45x) ❌
qps	42.912	90.16 (0.48x) ❌





Parallelize LLVM Pipeline

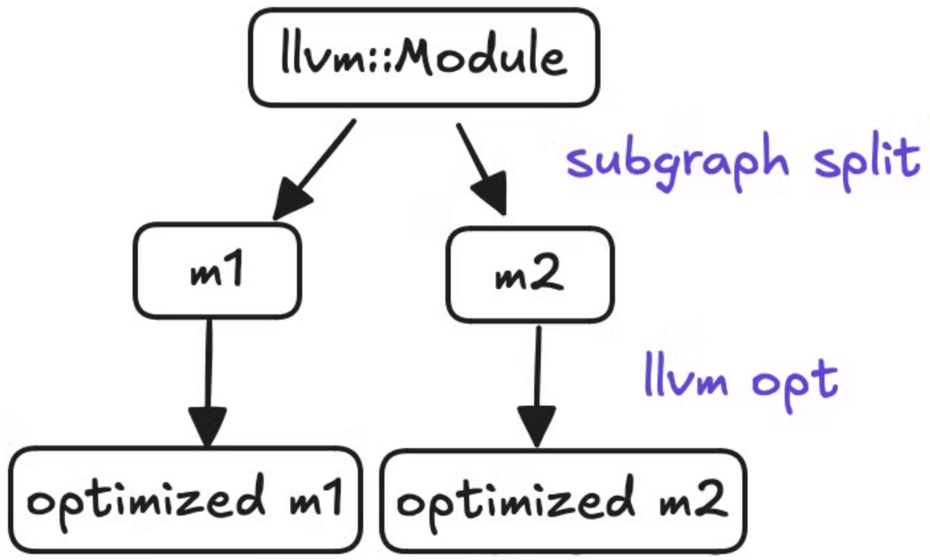
- Split llvm::Module into multiple submodules.
- Run each module split with separate LLVM pipeline in parallel.
- Two-level splitting:
  - Subgraphs with full function callgraph to run LLVM optimization pipeline including the inliner.
  - Subgraph further into functions for llc* pipeline (codegen) in parallel.
- Put codegen result of each split together to generate the output binary.

Parallelize LLVM Pipeline

llvm::Module

```
; llvm::Module
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

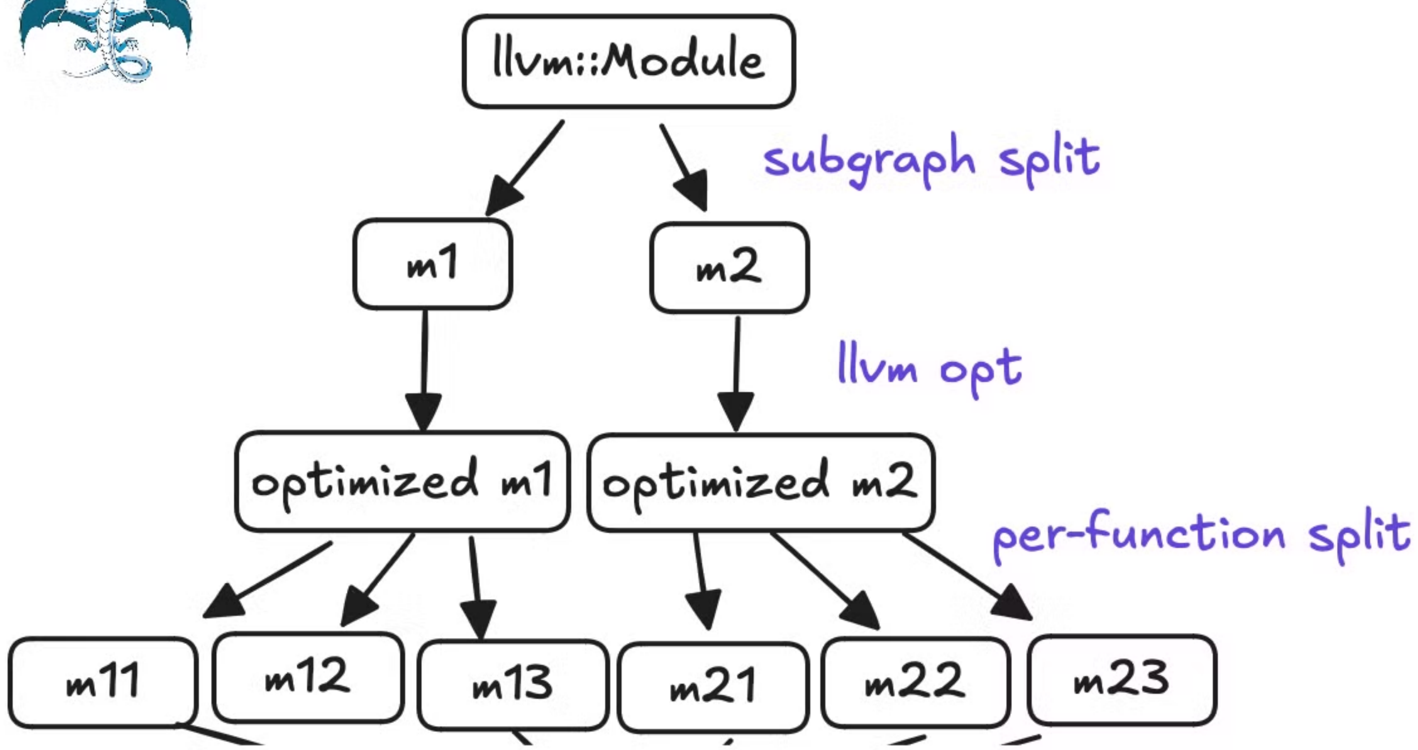
Parallelize LLVM Pipeline




```
; llvm::Module - m1
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

```
; llvm::Module - m2
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}
define internal void @g() #1 {
  ret void
}
define internal void @h() #1 {
  ret void
}
```

Parallelize LLVM Pipeline




```
; llvm::Module - m11
define dso_local void @foo() #0 {
  call void @g()
  call void @h()
  ret void
}

declare void @g() #1
declare void @h() #1
```

```
; llvm::Module - m21
define dso_local void @bar() #0 {
  call void @g()
  call void @h()
  ret void
}

declare void @g() #1
declare void @h() #1
```

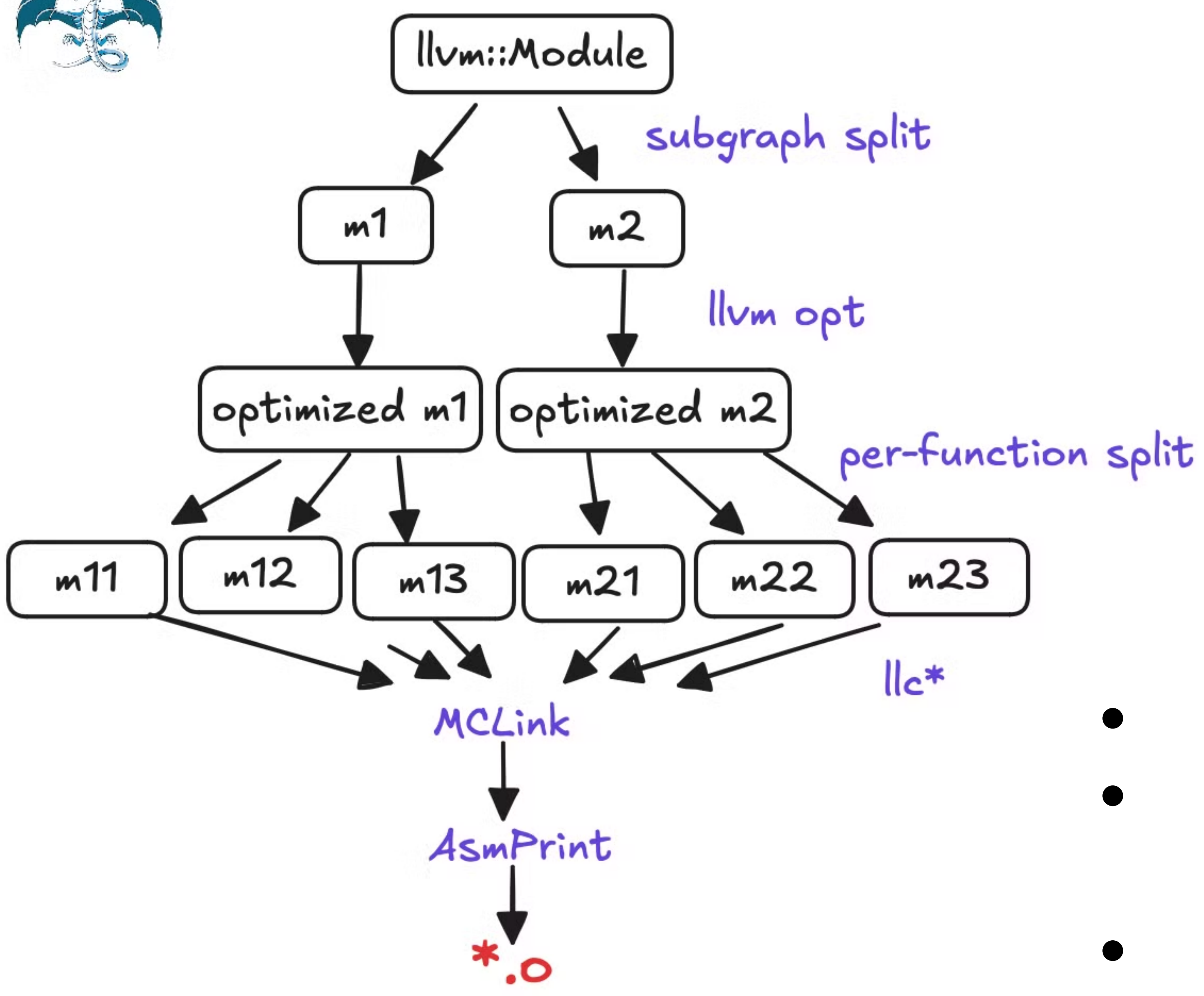
```
; llvm::Module - m12
define weak void @g() #1 {
  ret void
}
```

```
; llvm::Module - m22
define weak void @g() #1 {
  ret void
}
```

```
; llvm::Module - m13
define weak void @h() #1 {
  ret void
}
```

```
; llvm::Module - m23
define weak void @h() #1 {
  ret void
}
```

Parallelize LLVM Pipeline




```
SYMBOL TABLE:
0000000000000000 l    F __TEXT,__text ltmp0
000000000000000c l    F __TEXT,__text l_register_call_dtors.0
0000000000000038 l    F __TEXT,__text l_call_dtors.0
0000000000000050 l    O __DATA,__mod_init_func ltmp1
0000000000000058 l    O __LD,__compact_unwind ltmp2
00000000000000d8 l    O __TEXT,__eh_frame ltmp3
...
0000000000000000 g    F __TEXT,__text _foo
0000000000000004 g    F __TEXT,__text _bar
0000000000000000        *UND* __cxa_atexit
0000000000000000  w     *UND* __dso_handle
```

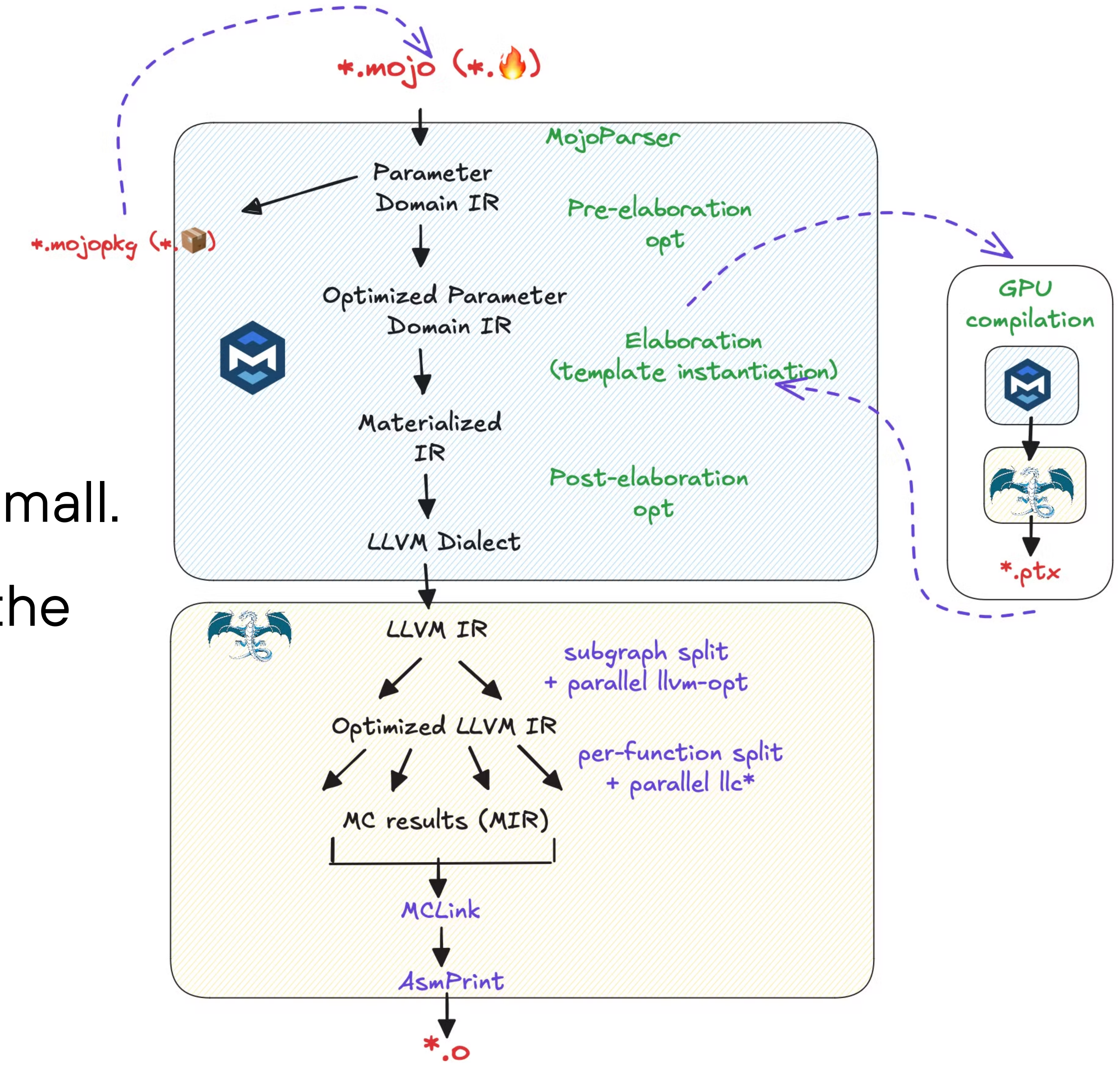
- **llc*** - Run codegen (llc) and stop before AsmPrint.
- **MCLink**- linking codegen results in memory: reduction for ConstantPool ids, Global MCSymbols, etc.
- AsmPrint linked result into one object file.
- We'd like to upstream this!

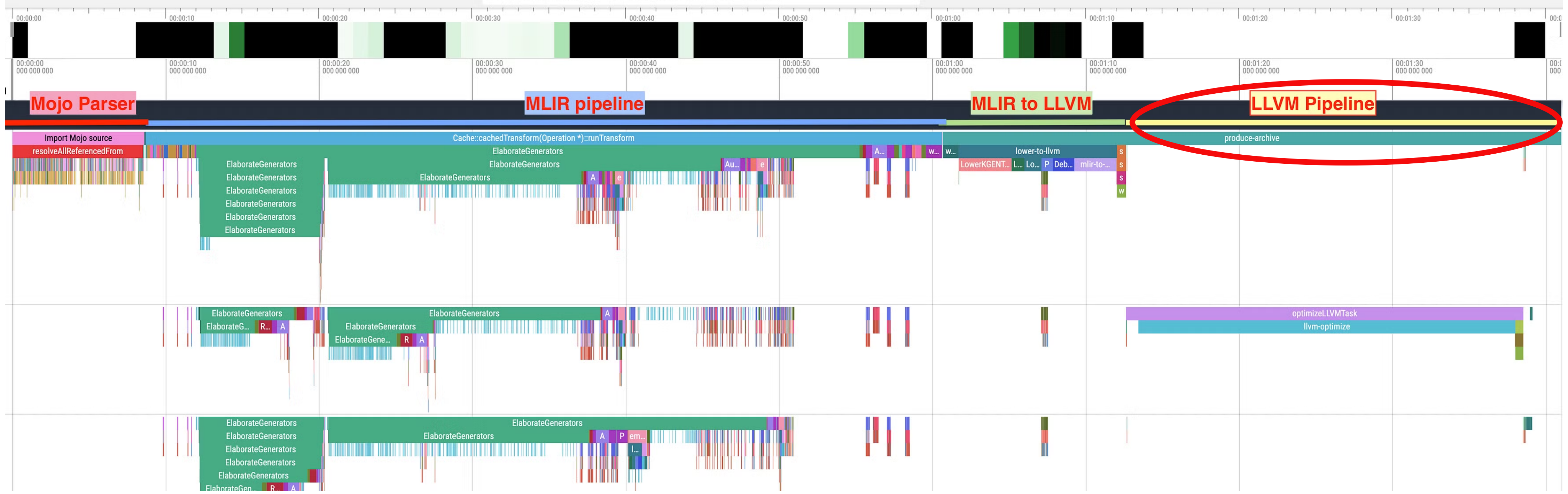
Infrastructure Challenges

- LLVM is not thread-safe
  - Separate copies of llvm::LLVMContext, llvm::MCContext for each split to compile in parallel.
  - Using llvm bitcode to serialize and deserialize IR for parallel splitting and compilation.
- Inefficiency in llvm bitcode:
  - Encodes each char as uint64_t in a string (8x space needed).
  - Bitcode reader performs 3 ultimate copies for strings.
  - llvm::getLazyBitcodeModule only materialize function bodies lazily but not constants which are fully parsed (PTX has large constants with O0).
- Linking codegen results:
  - Linking multiple parallel splits need to be put all in the same llvm::LLMContext.
  - TargetMachine is not stateless which increases memory footprint with per-split copies.
  - Target specific backends are private APIs and hard to change/access off-tree.

Conclusions

- Compilation time is a function of IR input size to LLVM instead of LLVM optimizing for performance.
- Moving optimization passes early in MLIR
  - Progressive optimization, choking the funnel to keep IR size small.
  - Mojo's high-level IR representation helps to reduce some of the inherent algorithmic complexity for optimizations.
  - Leverage framework and intra-pass parallelization.
- Using LLVM unconventionally
  - Two-level parallelization
  - MC-level linking
  - 1 mojo => 1 object file
- Significantly cut down LLVM pipeline time for overall mojo compilation (60-80% to 20-30%).
- Leverage the best of MLIR and LLVM.





Questions

Acknowledgement:
The Mojo Language Team at Modular