Modular

020C13

Mojo 🔥

A system programming language for heterogenous computing

Mojo 🔥 at a glance

Pythonic system programming language

- Driving SoTA in compiler and language design
- Forget everything you know about Python! :-)

One year old and still in development

- Freely available on Linux, Mac and Windows
- Full LLVM-based toolchain + VSCode LSP support
- Full emoji file extension support

Launched in May, already growing a vibrant community:

- 150K users overall, 22K+ users on Discord

Well funded, long term commitment

```
fn mandelbrot_kernel[
    simd_width: Int
](c: ComplexSIMD[float_type, simd_width]) ->
    SIMD[float_type, simd_width]:
    """
    A vectorized implementation of the
    inner mandelbrot computation.
    """
    let cx = c.re
    let cy = c.im
    var x = SIMD[float_type, simd_width](0)
    var y = SIMD[float_type, simd_width](0)
    var y2 = SIMD[float_type, simd_width](0)
    var iters = SIMD[float_type, simd_width](0)

    var t: SIMD[DType.bool, simd_width] = True
    for i in range(MAX_ITERS):
        if not t.reduce_or():
            break
        y2 = y * y
        y = x.fma(y + y, cy)
        t = x.fma(x, y2) <= 4
        x = x.fma(x, cx - y2)
        iters = t.select(iters + 1, iters)
    return iters
```

Agenda

01	Why Modular, Why Mojo?
02	Mojo Design Approach
03	Mojo Internals 101
04	Mojo + Accelerated Compute
05	Looking ahead

Modular

Why Modular, Why Mojo🔥 ?

If AI is so important, why is all the software infrastructure so bad?

What's wrong with AI* Infrastructure?

Building and deploying models requires dozens of translators, deployment systems, quantization tools, vendor specific compilers and kernel libraries!

Why?

- No one has time to start from first principles
- Organizational politics / incentive structures
- Solving this is really hard!

We need fewer things, that work better!

*Note: We use "AI" as an abbreviation for "*distributed heterogeneous compute*" systems


Modular

Unify AI from the Bottom Up

A next generation "AI Engine" to unify the world

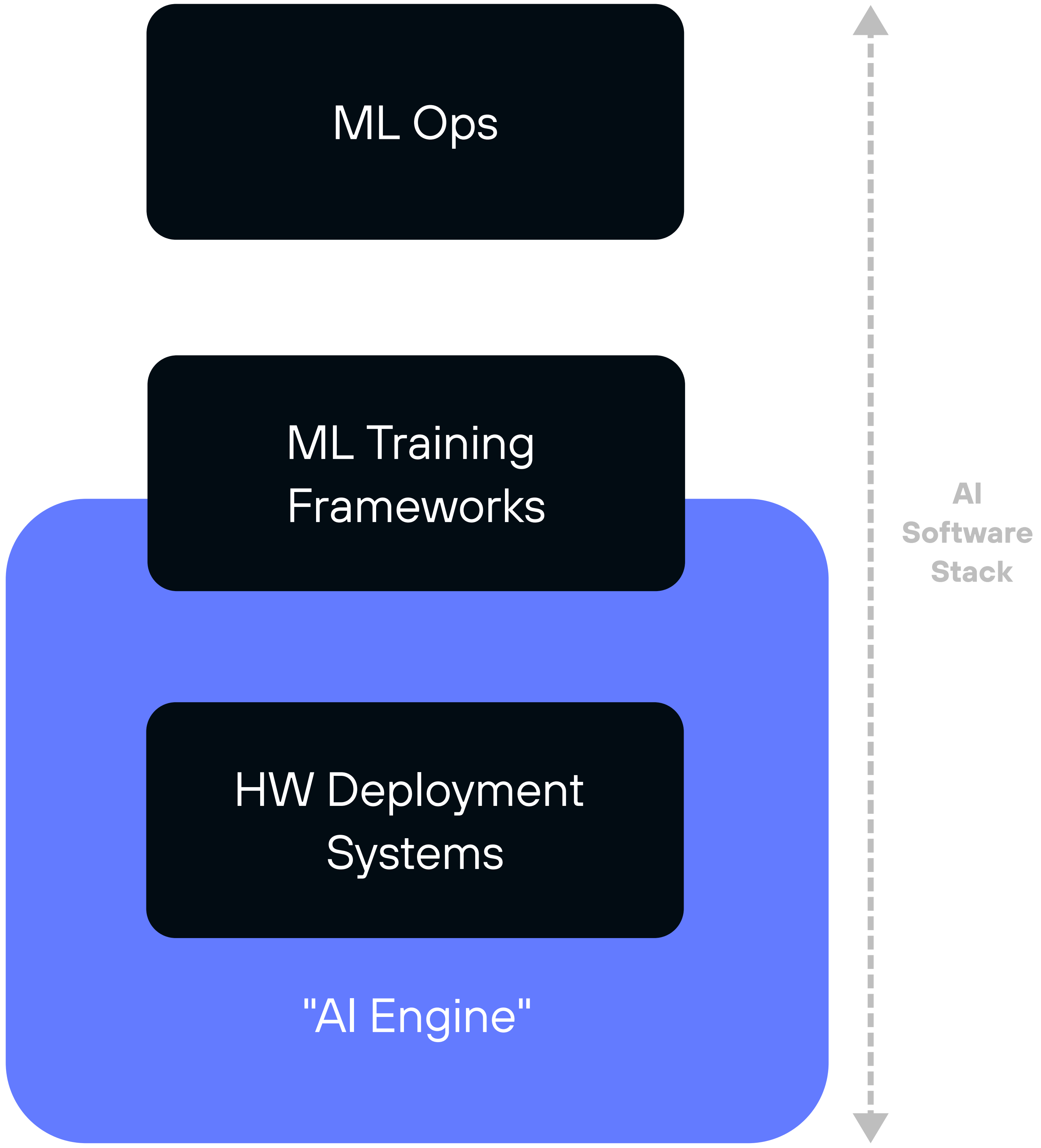
- Unify hardware, algorithms, and frameworks
- We've been on this quest for many years!

Meet AI developers where they are

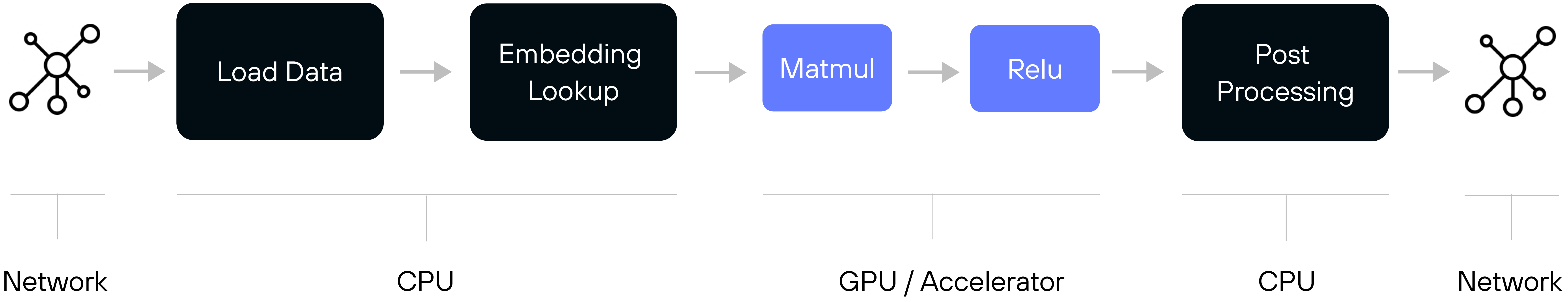
- Drop in compatible with PyTorch, JAX, and TensorFlow
- Few AI devs want to rewrite their models

Not a research project

- Much has been learned over the last ~8 yrs of AI infra
- Bring best-in-class techniques into one system
- First principles design + aligned team of experts



What is an AI Engine?



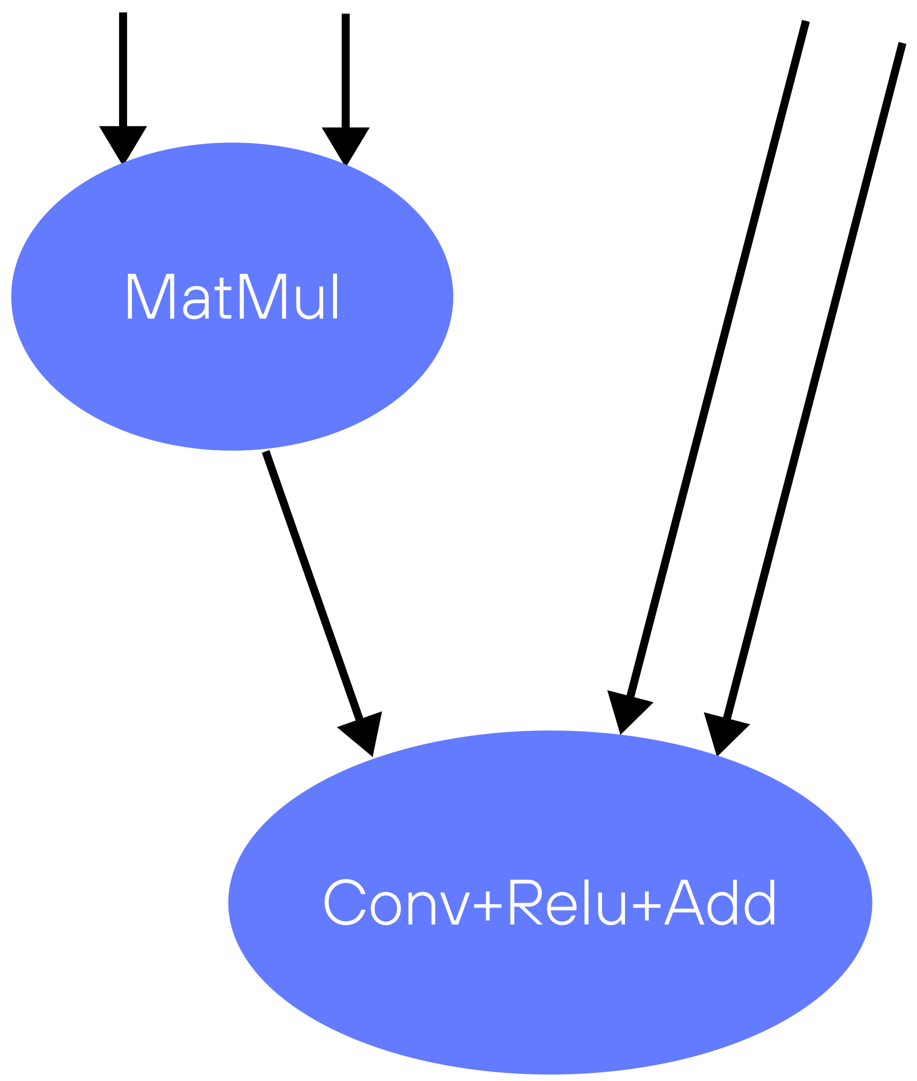
A declarative "operator graph" - sometimes small subgraphs

- enables transformation over the compute itself

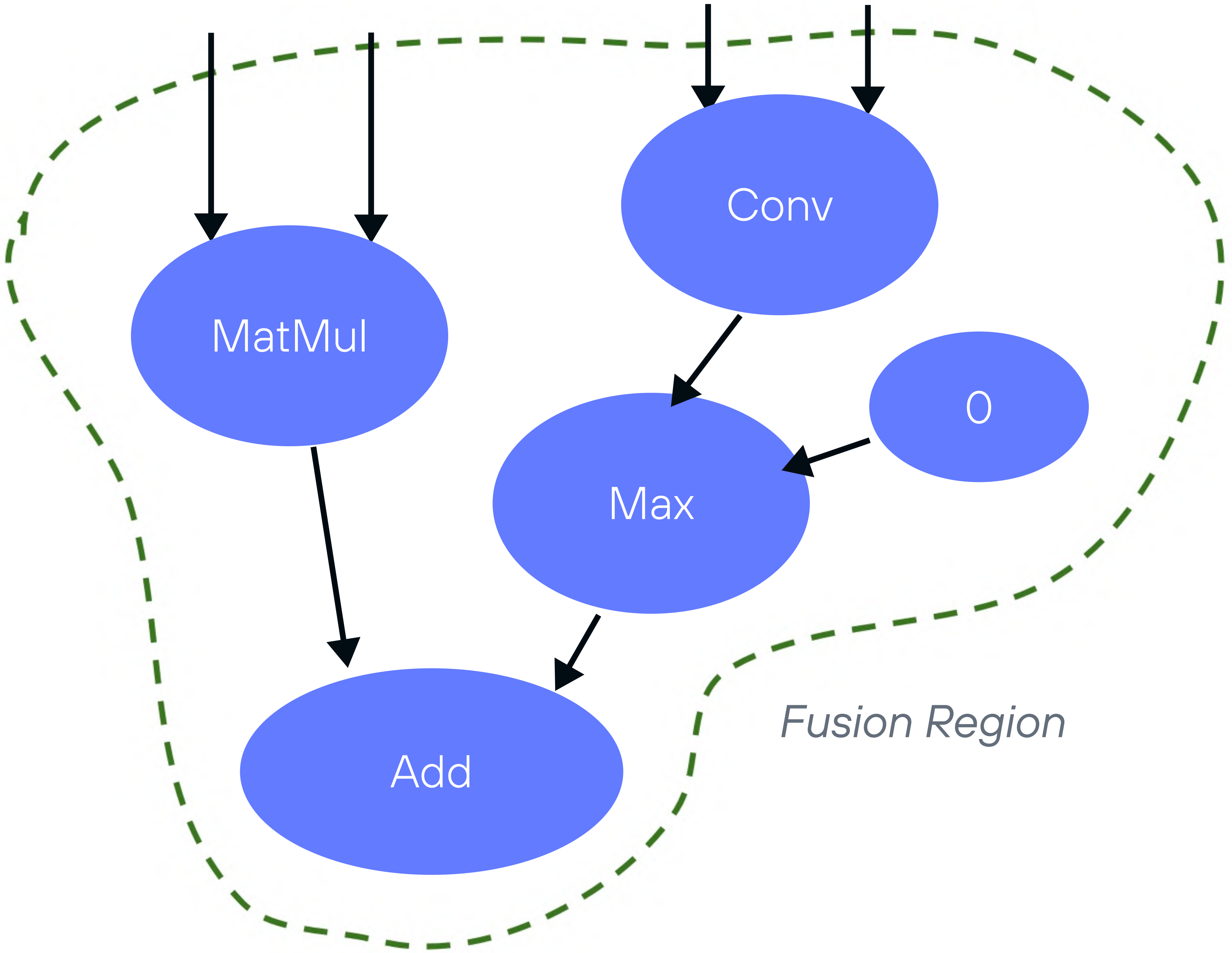
Manages distributed heterogeneous compute:

- This is more than just software for a single accelerator

AI Engine Evolution



Hand Coded Kernel Libraries



ML Compilers

Neither approach scales!

Challenges with ML Compilers

Generality!

Many common limitations…

- Dynamic shapes
- Sparsity
- Quantization
- Custom ops
- Embedded support
- Model coverage

Hard to invest in this when funded by HW enablement project:

- AI is an end to end parallel compute problem, not just an accelerated matmul problem
- Hardware-first software drives AI fragmentation

"Generality is, indeed, an indispensable ingredient of reality; for mere individual existence or actuality without any regularity whatever is a nullity. Chaos is pure nothing.

\- Charles Sanders Peirce

Community

Difficult to hire compiler engineers ...

- ... who have AI modeling experience, and
- ... who know exotic numerics, and
- ... who know specialized HW details

AI Research cannot rely on: "compiler engineer in-the-loop"!

Re-encoding all of computing into "IR Builders" doesn't scale

- We need to bring programmability back to AI!

Language + Developer Fragmentation

Model

System

Hardware

CUDA (and others)

How can you co-optimize host and accelerator code in different languages?

Modular

Mojo's Design Approach

Building a new language is a lot of work!

Initial goal: De-risk our core hypothesis

01

Prove we can beat SoTA kernels on a wide range of hardware

Demonstrate rapid architectural generality without performance loss

Both μbenchmark and end-to-end

02

Prove integration of novel next-generation compiler features

Metaprogramming, generalized fusion, autotuning, integrated caching, distributed compilation, unconventional use of LLVM, etc

03

For a de-risk, we don’t care about syntax!

Can *late bind* to EDSL, language, etc.

Many options exist if the core tech investment works out

"Compiler first" design approach

Build the compiler codegen strategy + unrelated parts of AI Engine

- Validated by writing MLIR directly, allowing us to iterate rapidly
- MLIR makes it very easy to prototype and build novel compilers

We succeeded!

- Beat SoTA kernel libraries / vendor compilers on key workloads
- Re-learned *how painful* it is to write large amounts of MLIR by hand

Time for Syntax! What approach?

We need to evaluate tradeoffs between:

- an **existing language** – e.g. C++ or Swift or Julia
- an **EDSL** in Python or C++
- a **new**, invented, language

Start from our goals:

- Enable usability, for our fancy compiler technology
- Meet AI devs where they are: **in Python** (doom voice)

Python drives the requirement: no C++/Swift/Julia/etc

Why not an Embedded DSL (EDSL) ?

Many EDSLs in Python & C++ exist, because:

- ☑ Much lower cost to produce than a full language
- ☑ Don’t need to implement language tooling
- ☑ Fast time to market

Challenges with EDSLs:

- ☑ Poor usability, poor tooling, poor debugging
- ☑ Can't extend/fix the host language

Our goals require full-stack innovation (including the host) and aim for best usability!

Embedded Domain Specific Language

A DomainSpecificLanguage that is defined as a library for a generic "host" programming language. The embedded DSL inherits the generic language constructs of its host language - sequencing, conditionals, iteration, functions, etc. - and adds domain-specific primitives that allow programmers to work at a much higher level of abstraction. Multiple EDSLs can easily be combined into a single program and a programmer can use the facilities of the host language to extend the existing DSLs or use them to build an even higher level DSL.

Build a new language?

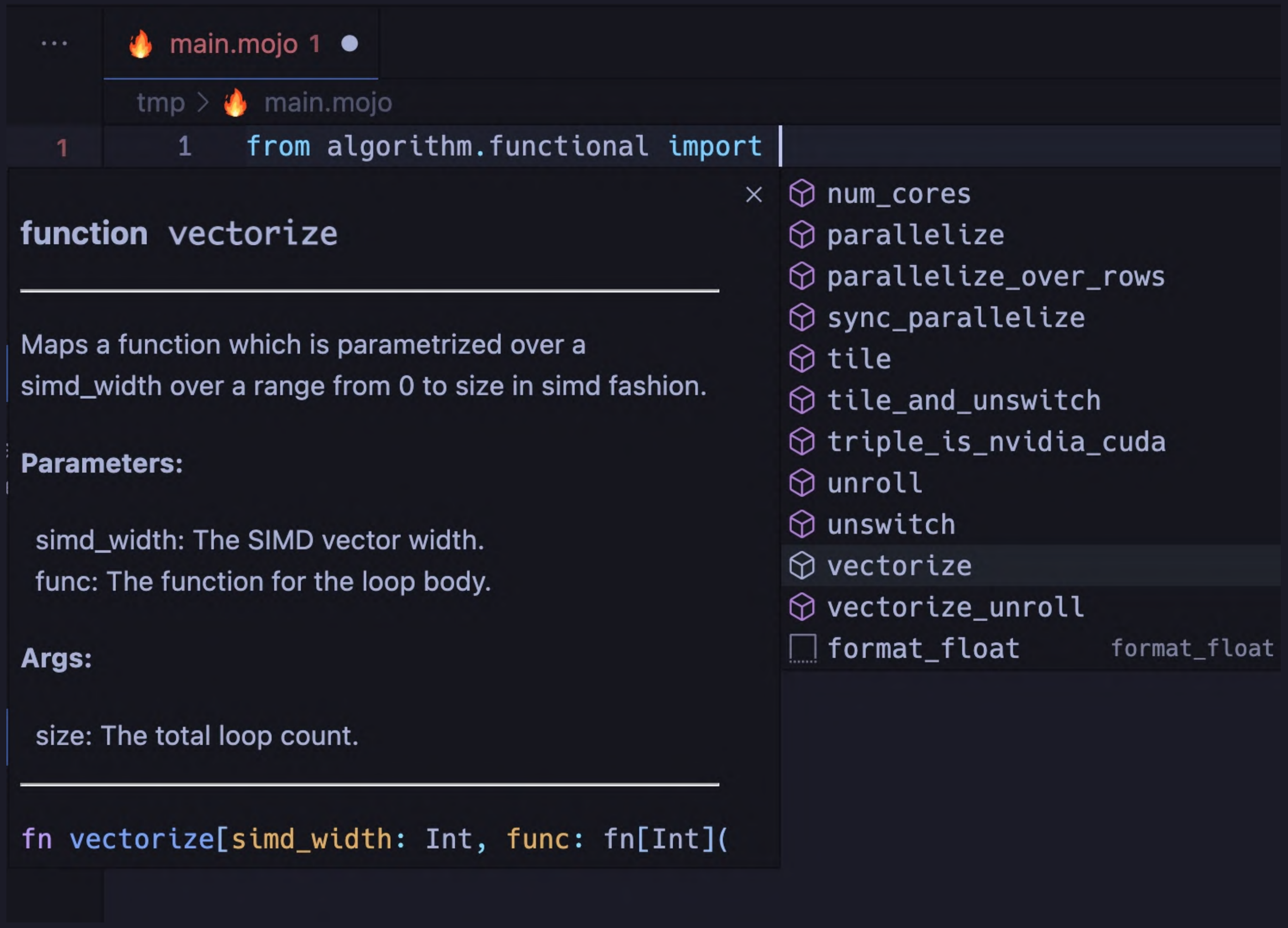
Only way to deliver the *best quality result*

- A native tools experience, debugger etc
- Full generality for host CPUs: Python won't cut it

However, this requires:

- Consistent vision
- Long term commitment
- Funding for the development
- Ability to attract specialized talent
- Big target market of developers

Ridiculously expensive to do right!



Mojo🔥 provides full VSCode / LSP support, REPL, Jupyter, and (shipping soon) LLDB Debugger

Build a new language!

Only way to deliver the *best quality result*

- AI developers are really important to the world
- We're tired of point solutions, research-quality tools, flashy demos that don't generalize

However, this requires:

✓ Consistent vision
✓ Long term commitment
✓ Funding for the development
✓ Ability to attract specialized talent
✓ Big target market of developers

We have done this before:

OpenCL Clang Swift

Mojo🔥 design points

01

Member of the Python 🐍 family

Give superpowers to Python coders

Will grow into a “Python++” superset over time (no “Python 4” fragmentation)

02

Focused on performance & systems programming

Work backward from unlocking HW - not forward from legacy Python

Anything with a program counter (PC)

03

Expose Modular’s next-generation compiler technology

Unlock the full power of MLIR

Fancy compiler tech like autofusion

Support the needs of the AI engine

Modular

Mojo🔥 Internals 101

Core elements of the language + compiler

A programming language ***for*** MLIR?

Computers are complicated!

Are type systems solved? Look at floating point!

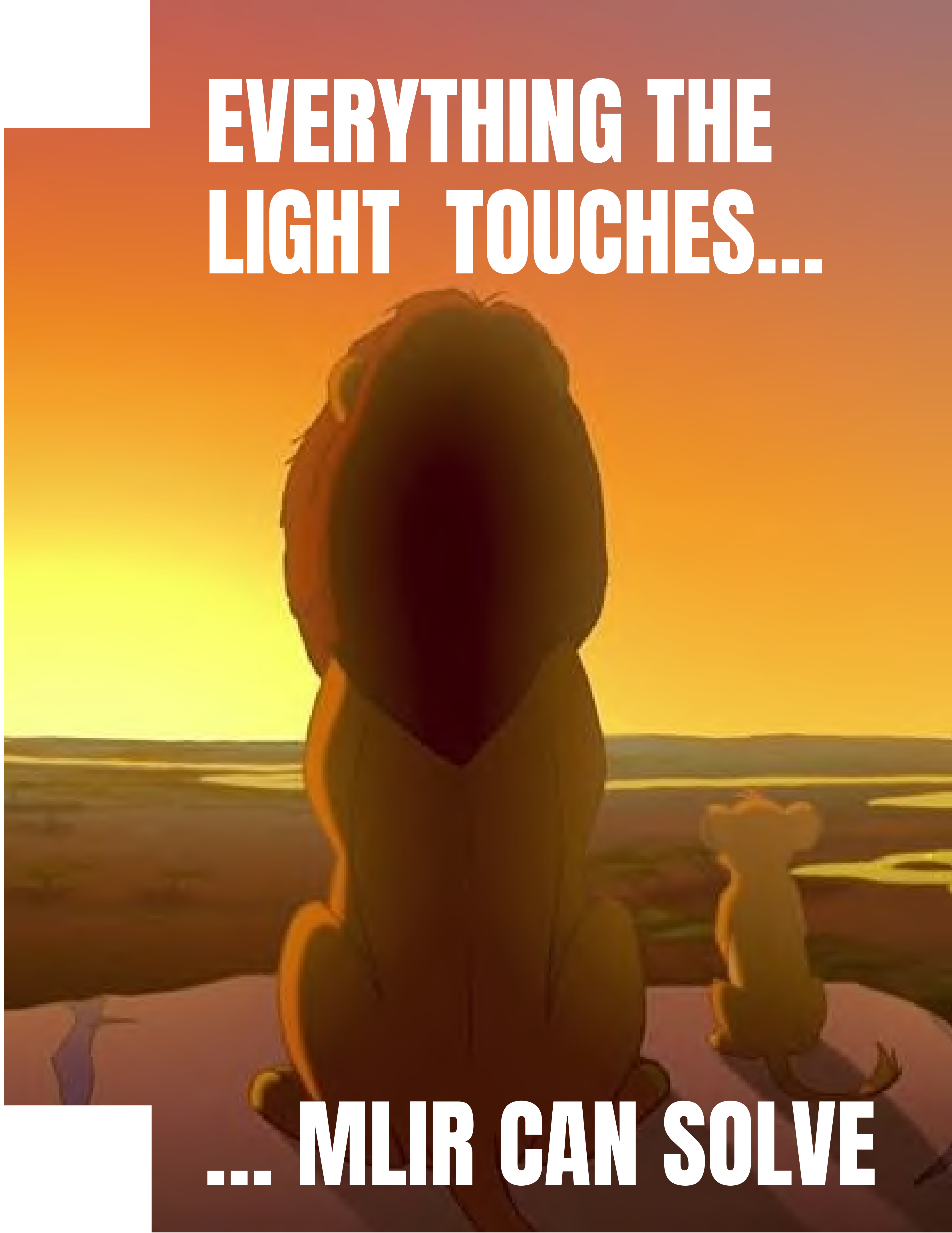
- F16, BF16, F32, F64, and maybe F80 ... right?

What about:

- Float8E5M2
- Float8E4M3FN
- Float8E5M2FNUZ
- Float8E4M3FNUZ
- Float8E4M3B11FNUZ!

What about tiled accelerators?

We need syntactic sugar for MLIR!



A library-first language 📚

C++ has an odd historical design

- `double` is built-in to language
- `std::complex` is a library

Goal: Push language design into libraries!

- Extend without changing the compiler
- Reduce engineering effort 👷🏽‍♀️
- Talk to all the weird hardware 🔨

A enormous opportunity!

Python 🐍 to the rescue!

```python
class Int:
    def __init__(self, value):
        self.value = value

    def __add__(self, rhs): ...

    def __lt__(self, rhs): ...
```

Syntactic sugar for MLIR

```
struct Int:
    var value: __mlir_type.index

    fn __add__(self, rhs: Int) -> Int:
        return __mlir_op.`index.add`(self.value, rhs.value)

    fn __lt__(self, rhs: Int) -> Bool:
        return __mlir_op.`index.cmp`[
              pred = __mlir_attr.`#index<cmp_pred slt>`
        ](self.value, rhs.value)
```

Zero cost abstractions

Trivial

- Bag of bits

`@register_passable`

- Lives in SSA registers

`@always_inline("nodebug")`

- No function call overhead
- No generated debug info

```
@register_passable("trivial")
struct Bool:
    var value: __mlir_type.i1

    @always_inline("nodebug")
    fn __and__(self, rhs: Bool) -> Bool:
        return __mlir_op.`arith.andi`(
            self.value, rhs.value)
```

Putting it together

```
var i = 0
while i < 10:
  print(i)
  i += 1
```

```
%i = lit.varlet.decl "i" : !lit.ref<mut !Int, *"`i0">
%0 = kgen.param.constant: !Int = <{value = 0}>
lit.ref.store %0, %i : <mut !Int, *"`i0">
```

```
hlcf.loop {
  %1 = lit.ref.load %i : <mut !Int, *"`i0">
  %2 = kgen.param.constant: !Int = <{value = 10}>
  %3 = kgen.call @Int::@__lt__(%1, %2)
  %4 = kgen.call @Bool::@__mlir_i1__(%3)
  hlcf.if %4 {
    hlcf.yield
  } else {
    hlcf.break
  }
  %5 = lit.ref.load %i : <mut !Int, *"`i0">
  kgen.call @print(%5)
  %7 = kgen.param.constant: !Int = <{value = 1}>
  kgen.call @Int::@__iadd__(%i, %7)
  hlcf.continue
}
```

```
%idx0 = index.constant 0
%idx10 = index.constant 10
%idx1 = index.constant 1
hlcf.loop (%arg2 = %idx0 : index) {
  %0 = index.cmp slt(%arg2, %idx10)
  hlcf.if %0 {
    hlcf.yield
  } else {
    hlcf.break
  }
  %1 = kgen.call @print(%arg2)
  %2 = index.add %arg2, %idx1
  hlcf.continue %2 : index
}
```

Mojo🔥

Language

Intermediate

Representation

Bring your own Dialect

Zero-cost MLIR wrappers form bottom layer of Mojo 🔥

Syntactic sugar 🍭 for MLIR

- Reusable MLIR front-end

```
struct Shape:
    var value: __mlir_type.`!mosh.ape`

    fn __add__(self, rhs: Self) -> Self:
        return __mlir_op.`mosh.concat`(
            self.value, rhs.value)

    fn __getitem__(self, n: Int) -> Int:
        return __mlir_op.`mosh.get_dim`(
            self.value, n.value)
```

EDSLs in Mojo for MLIR dialects!

```
fn matmul_like_fw(sh_a: Shape, sh_b: Shape)
           -> Shape:
    return sh_a.slice(0, -2) +
           Shape(sh_a[-2], sh_b[-1])
```

```
kgen.generator @matmul_like_fw(
    %arg0: !mosh.ape, %arg1: !mosh.ape)
    -> !mosh.ape {
  %idx-1 = index.constant = -1
  %idx0 = index.constant = 0
  %idx-2 = index.constant = -2
  %0 = mosh.slice(%arg0)[%idx0, %idx-2]
  %1 = mosh.get_dim(%arg0)[%idx-2]
  %2 = mosh.get_dim(%arg1)[%idx-1]
  %3 = mosh.new(%1, %2)
  %4 = mosh.concat(%0, %3)
  kgen.return %4 : !mosh.ape
}
```

EDSLs in Mojo for MLIR dialects!

```
fn matmul_like_fw(sh_a: Shape, sh_b: Shape)
            -> Shape:
    return sh_a.slice(0, -2) +
```

```
kgen.generator @matmul_like_fw(
    %arg0: !mosh.ape, %arg1: !mosh.ape)
    -> !mosh.ape {
  %idx-1 = index.constant = -1
  %idx0 = index.constant = 0
                                   %idx-2]
                                   2]
                                   1]
  %3 = mosh.new(%1, %2)
  %4 = mosh.concat(%0, %3)
  kgen.return %4 : !mosh.ape
}
```

Bonus: all the language tooling just works

Compile Time Metaprogramming

Mojo 🔥 needs ...

Hardware generality / single-source-of-truth

Kernel parameterization over vector length, unroll factor, tile factor, ...

C++ templates?

- Meta-lang != actual lang 😵‍💫
- Bad error messages 🤬
- Not powerful enough 😫

```
kgen.generator @microkernel<width>(
    %x: !pop.simd<f32, width>) -> !pop.simd<f32, width> {
  ...
}

kgen.generator @kernel(
    %in: !kgen.pointer, %out: !kgen.pointer,
    %size: index) {
  kgen.param.search width = <[2, 4, 8, 16, 32]>
  %step = kgen.param.constant = <width>
  scf.for %i = 0 to %size step %step {
    %x = pop.simd_load %in[%i] : <f32, width>
    %0 = kgen.call @microkernel<width>(%x)
    pop.simd_store %0 to %out[%i] : <f32, width>
  }
  kgen.return
}
```

🤔

Mojo 🔥 needs ...

... what Python 🐍 has

Powerful metaprogramming:

- Decorators
- Metaclasses
- Reflection

But ... Runtime based is slow - it will never run on the accelerator!

💡 Let's do it at compile time! 💡

Mojo Parameter Syntax

```
# Struct with parameters
struct SIMD[dtype: DType, width: Int]:
    ...


# "alias" declaration -> parameter
alias Float32 = SIMD[DType.f32, 1]
```

```
# Bind function parameters to type
fn first_class_simd[width: Int](
    x: SIMD[DType.float32, width]):
  pass
```

~= C++ templates

Meta-language = actual language

01

Mojo's metaprogramming language is just Mojo 🔥

02

Almost any user-defined type can be used at compile time

03

MLIR interpreter with memory model for compile-time code evaluation

MLIR interpreter for a stack-based programming language

(Tuesday's MLIR workshop)

Function can be called at either compile or run time

```
fn fill(lb: Int, ub: Int) -> Vector[Int]:
    var values = Vector[Int]()
    for i in range(lb, ub):
        values.append(i)
    return values
```

Vector with heap allocation

Vector computed at compile-time...
used at runtime!

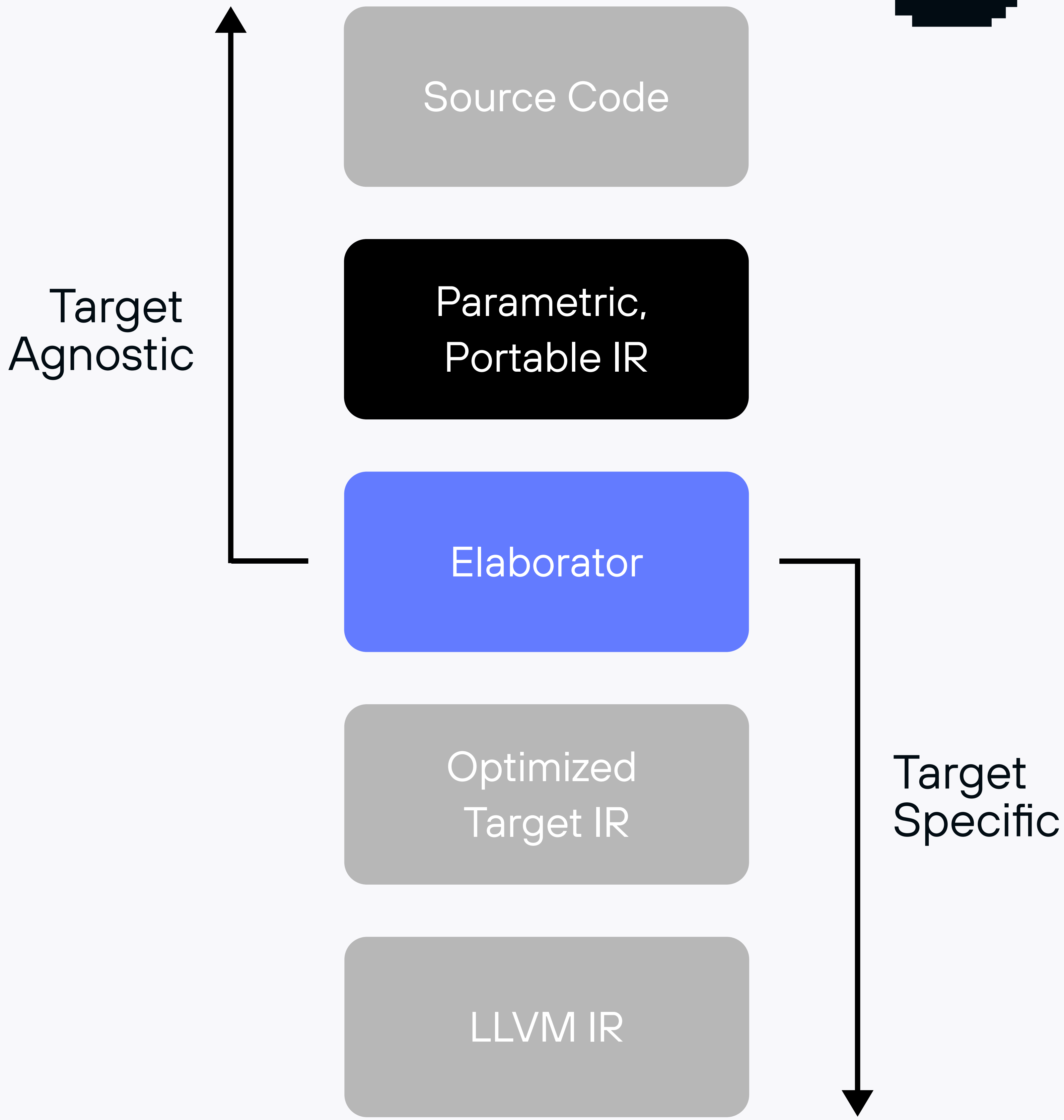
```
fn comptime_vector():
    alias vec = fill(15, 20)
    for e in vec: print(e)
```

Mojo 🔥 does not "instantiate" in its parser!

```
fn print_int[value: Int]():
    print(value)
```

↓

```
kgen.generator @print_int<value>() {
  %0 = kgen.param.constant = <value>
  kgen.call @print(%0)
  kgen.return
}
```



Elaboration Pass

```
kgen.generator @main() {
  kgen.call @print_int<42>()
  kgen.call @print_int<2023>()
}
```

```
kgen.func @main() {
  kgen.call @"print_int,value=42"()
  kgen.call @"print_int,value=2023"()
}
```

```
kgen.func @"print_int,value=42"() {
  %0 = kgen.param.constant = <42>
  kgen.call @print(%0)
}
```

```
kgen.func @"print_int,value=2023"() {
  %0 = kgen.param.constant = <2023>
  kgen.call @print(%0)
}
```

Autotuning!

```
# Vector-length agnostic function...
fn microkernel[width: Int](x: SIMD[DType.f32, width])
    -> SIMD[DType.f32, width]): ...

fn kernel(in: ..., out: ..., size: Int):
    # Best vec length? Let Mojo decide!
    alias width = autotune(2, 4, 8, 16, 32)
    for i in range(0, size, width):
        microkernel(in.simd_load[width](i))
```

Performance problems with C++ templates

```
template<typename T>
T add(const T &lhs, const T &rhs) {
  return lhs + rhs;
}
```

Passing by **const&** for generality

```
HeavyString add(const HeavyString &lhs,
                const HeavyString &rhs) {
  return lhs + rhs;
}
```

```
int add(const int &lhs, const int &rhs) {
  return lhs + rhs;
}
```

```
int x = ...
int y = ...
z = add(x, y);
```

Bad for performance for trivial types!

(When not inlined)

```
%1 = alloca i32
%2 = alloca i32
store i32 %x, i32* %1
store i32 %y, i32* %2
%z = call i32 @_Z3addRKiS0_(i32* %1, i32* %2)
```

Trivial arguments pinned to the stack

Late ABI Lowering

```
fn add[T: Addable](
    lhs: T, rhs: T) -> T:
return lhs + rhs
```

```
kgen.func @"add,T=String"(
    %out: !kgen.pointer<!String>
    %lhs: !kgen.pointer<!String>,
    %rhs: !kgen.pointer<!String>) {
  kgen.call @String::@__add__(
      %out, %lhs, %rhs)
}

kgen.func @"add,T=Int"(
    %lhs: index, %rhs: index) -> index {
  %0 = index.add %lhs, %rhs
  kgen.return %0 : index
}
```

@register_passable types are promoted during elaboration!

- Dovetails with borrow conventions

Mojo 🔥 CodeGen Architecture

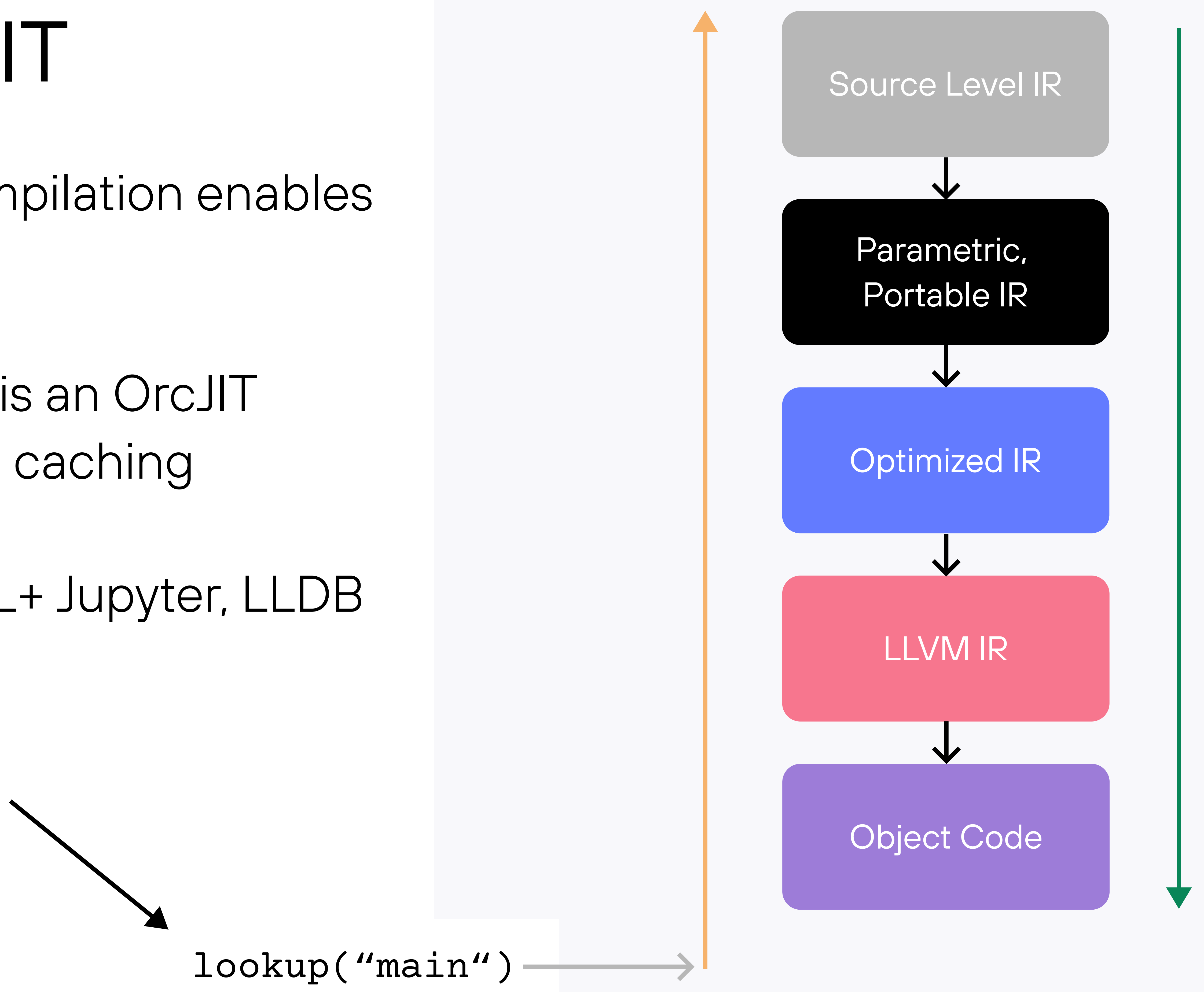
Driven by OrcJIT

Lazy demand-driven compilation enables responsive tooling

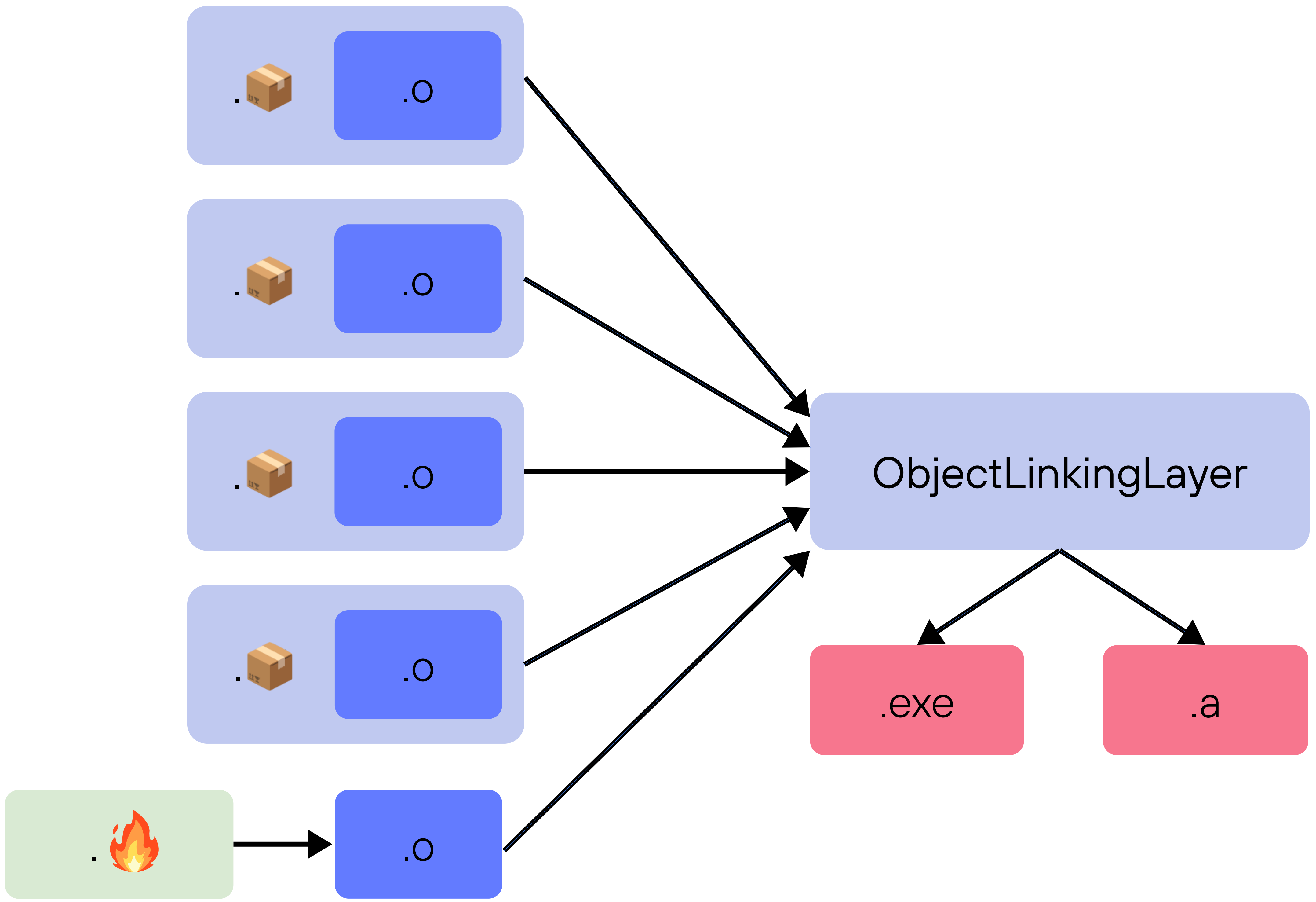
Each compilation phase is an OrcJIT materialization layer with caching

Powers autotuning, REPL+ Jupyter, LLDB exprs eval

E.g. mojo run my_file.🔥



OrcJIT … as a static archive generator

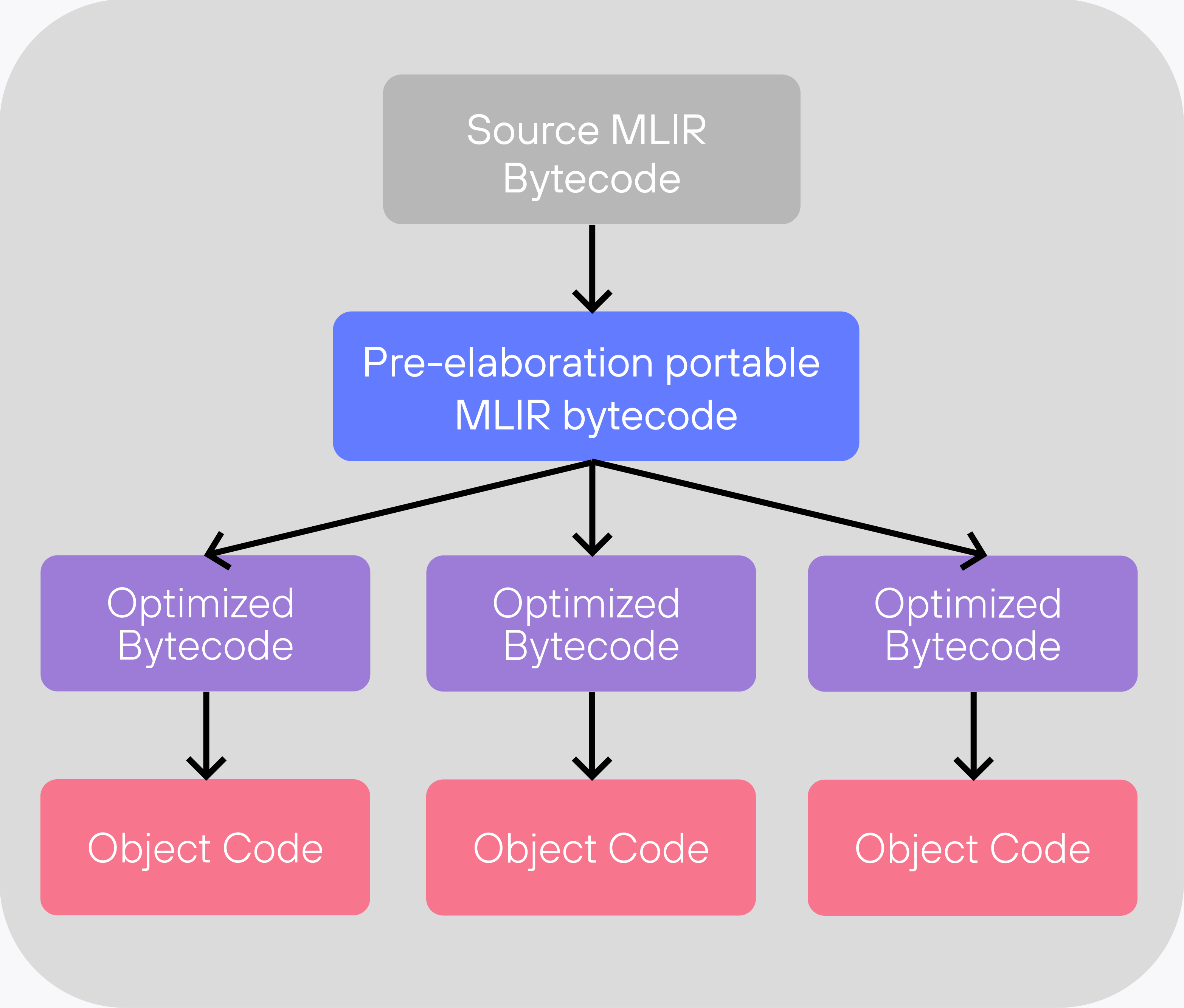


Architecturally portable code 📦

Mojo 🔥 can ship portable IR in packages without source code!

- Parametric bytecode is a much better "precompiled header"

Packages may optionally contain target-specific IR and "fat" object code for multiple targets

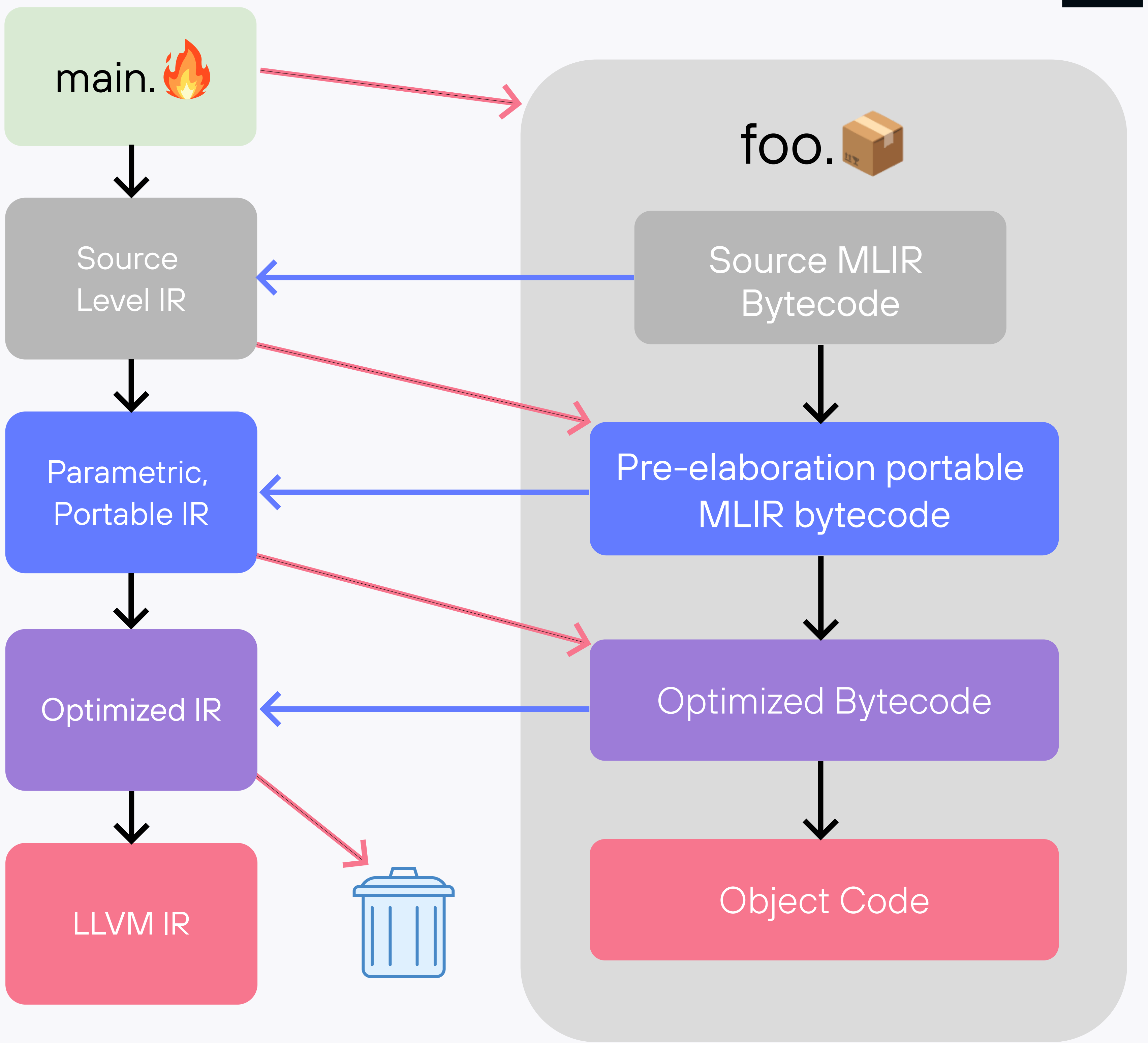


Compilation with Packages

```
from foo import bar

fn main():
    bar()
```

At each phase, pull in the pre-processed IR instead of re-running passes.

Optimized IR from package is tossed before LLVM lowering



LLVM IR, used unconventionally 😏

We love 🐉, but the LLVM optimizer… has problems

Single-threaded LLVM IR optimizer

- 100x slowdown on emerging / modern machines

Weak and unpredictable loop optimizer

- High performance relies on control and predictability
- Want to autotune loop optimization parameters

Some stuff built for Clang🔔 doesn't apply to Mojo🔥

Good news! to the rescue!

```
fn kernel[vec_len: Int](
    in: ..., out: ..., size: Int):

    # Autotune the unroll factor!
    alias factor = autotune(1, 2, 4)

    @unroll(factor)
    for i in range(0, size, vec_len):
        ...
```

LLVM … the good parts

LLVM is good for:

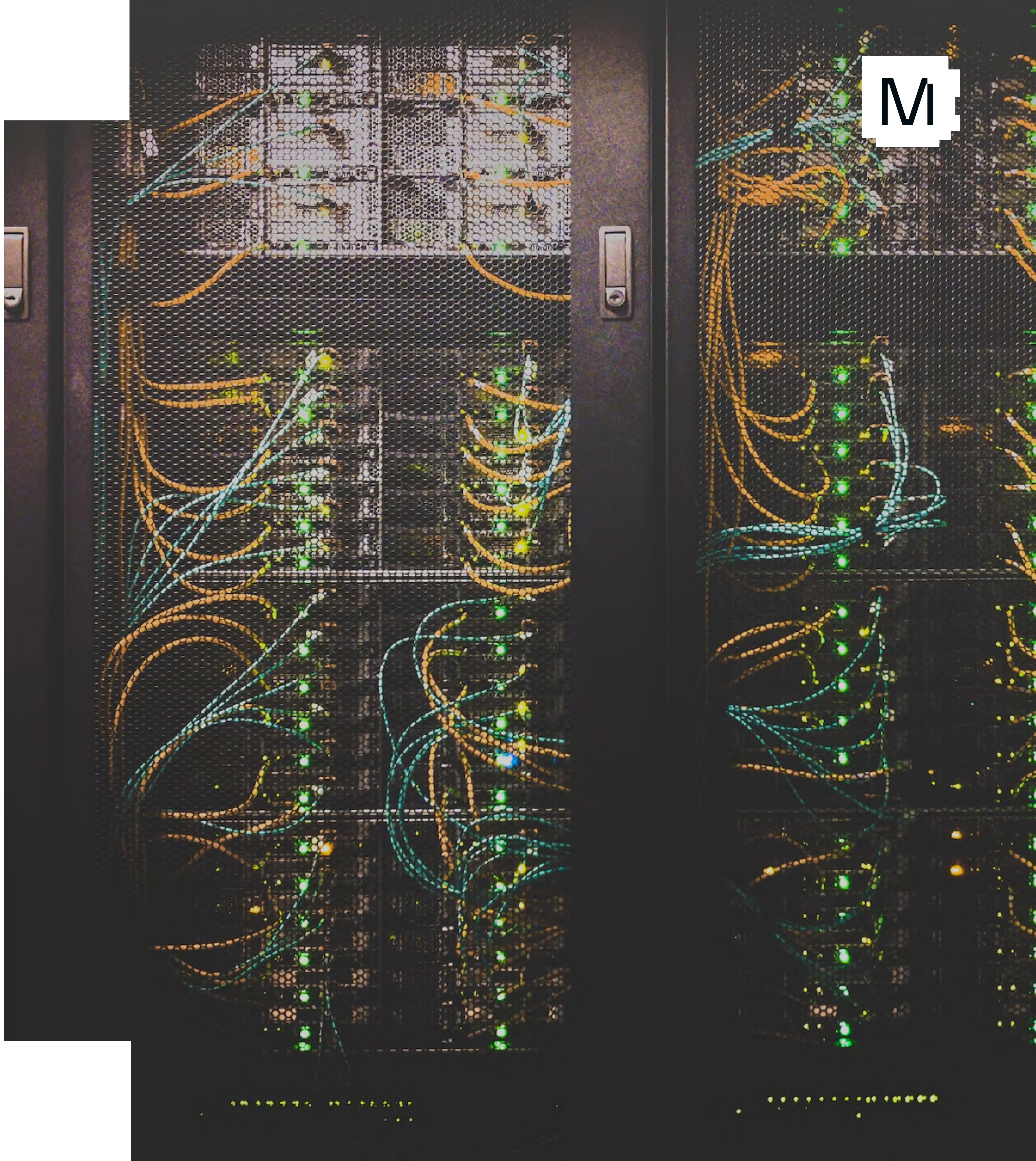
- GVN, Load/Store Optimization, LSR, etc
- scalar optimization (e.g. instcombine)
- target-specific code generation

We need to disable:

- Vectorizer, loop unroller, etc
- Inliner and other IPO passes

Solution: replace these!

- Build new MLIR passes
- Replace others with Mojo libraries



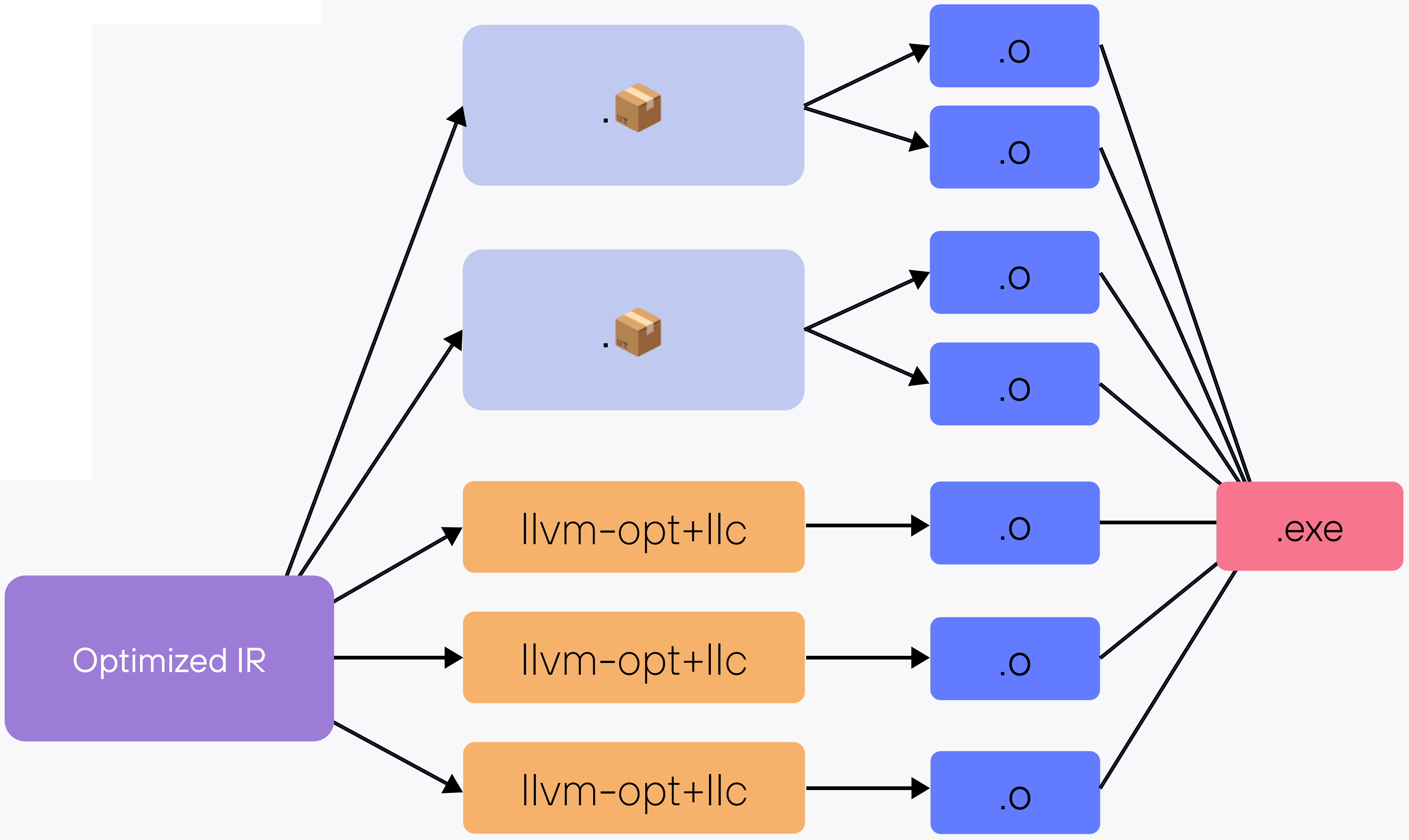
LLVM as a per-function code generator!

New MLIR passes

- Fast, parallel, controlled
- Parameterized / elaboratable

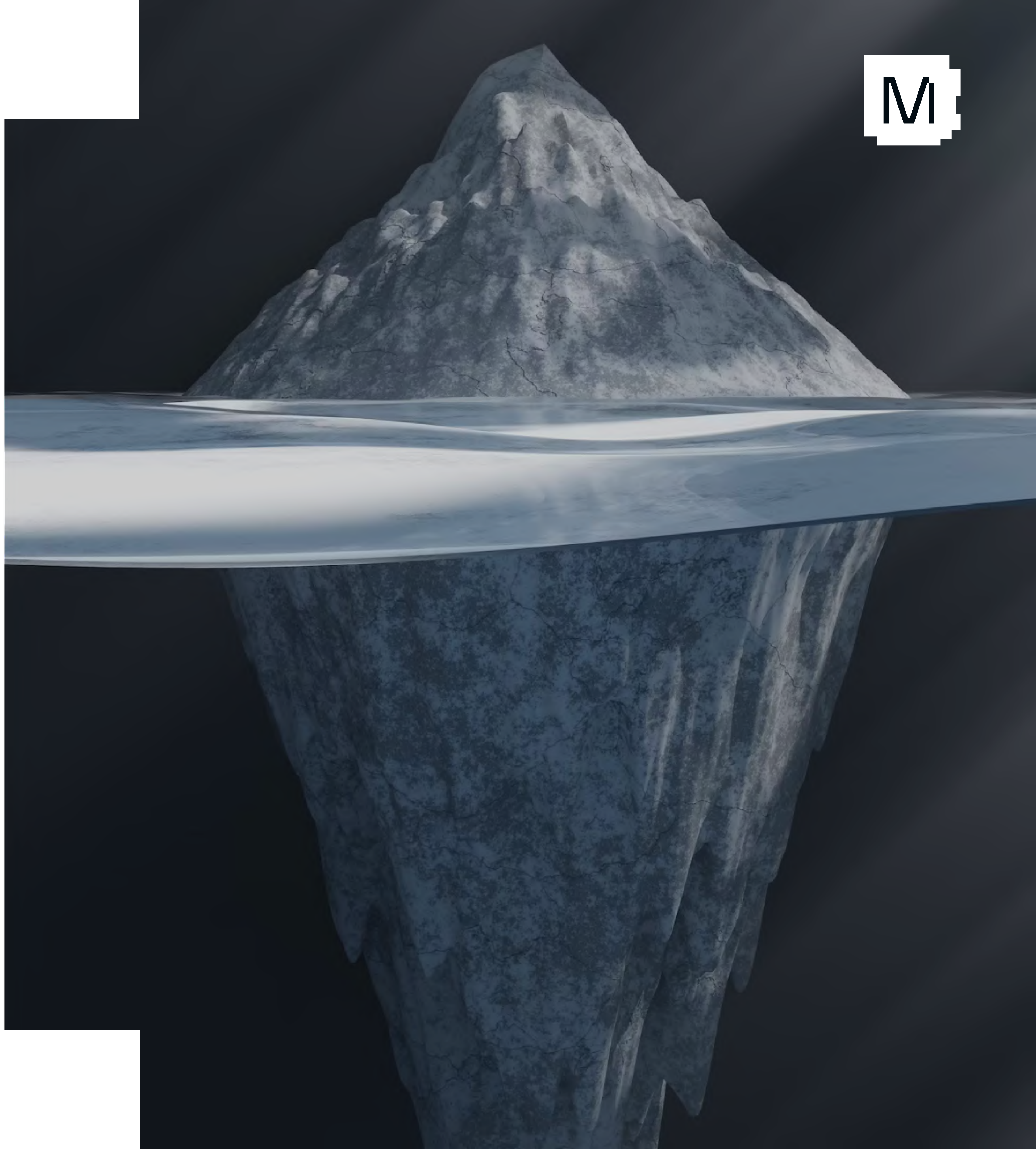
One LLVMContext per-function

- Parallelism!
- Easy caching!



So much more …

- CPython interoperability
- Parameter design in MLIR
- Lifetimes, ownership and early destruction
- Keyword arguments and parameters
- Function auto-parameterization
- @value decorator and value semantics
- Cross compilation, GPU programming
- REPL and Jupyter notebook
- LSP server, vscode plugin, code completion
- First class LLDB integration
- Compile time IR reflection
- Mojo Concurrency model
- Traits and static polymorphism
- …



Modular

Mojo🔥 for High Performance

The need for speed

A look at existing performance libraries


Whatever it takes for performance
... at the cost of suffering for performance engineers

Write in Assembly!

Please, no...

```
lea       rax,[rdx+r8*2]
vpmovzxbw ymm4,XMMWORD PTR [rdx]
vpmovzxbw ymm5,XMMWORD PTR [rdx+r8]
vpmovzxbw ymm6,XMMWORD PTR [rax]
vpmovzxbw ymm7,XMMWORD PTR [rax+r8]
lea       rax,[rcx+r11*4]
vmovdqu  YMMWORD PTR [rcx],ymm4
vmovdqu  YMMWORD PTR [rcx+r11*2],ymm5
vmovdqu  YMMWORD PTR [rax],ymm6
vmovdqu  YMMWORD PTR [rax+r11*2],ymm7
vpaddw   ymm0,ymm0,ymm4
vpaddw   ymm1,ymm1,ymm5
vpaddw   ymm2,ymm2,ymm6
vpaddw   ymm3,ymm3,ymm7
add      rdx,16
add      rcx,16*2
sub      rbx,16
```

C++ Templates

```cpp
static constexpr auto GemmDefault =
    ck::tensor_operation::device::GemmSpecialization::Default;

using DeviceGemmInstance = ck::tensor_operation::device::DeviceGemmXdl<
    ADataType, BDataType, CDataType, AccDataType, ALayout, BLayout, CLayout,
    AElementOp, BElementOp, CElementOp, GemmDefault, 256, 128, 128, 4, 2, 16,
    16, 4, 4, S<4, 64, 1>, S<1, 0, 2>, S<1, 0, 2>, 2, 2, 2, true, S<4, 64, 1>,
    S<1, 0, 2>, S<1, 0, 2>, 2, 2, 2, true, 7, 1>;

using ReferenceGemmInstance =
    ck::tensor_operation::host::ReferenceGemm<ADataType, BDataType, CDataType,
                                              AccDataType, AElementOp,
                                              BElementOp, CElementOp>;

#include "run_gemm_example.inc"
```

Source: Composable Kernels

C++ DSL for ASM

```
L(labels[4]);
test(K, 2);
jle(labels[5], T_NEAR);
innerkernel2(unroll_m, unroll_n, isLoad1Unmasked, isLoad2Unmasked, isDirect,
             isCopy, useFma, reg00, reg01, reg02, reg03, reg04, reg05,
             reg06, reg07, reg08, reg09, reg10, reg11, reg12, reg13, reg14,
             reg15, reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23);
align(16);

L(labels[5]);
if (unroll_m == 16) {
    if (unroll_n <= 3) {
        vaddps(reg00, reg00, reg12);
        vaddps(reg01, reg01, reg13);
        vaddps(reg02, reg02, reg14);
        vaddps(reg06, reg06, reg18);
        vaddps(reg07, reg07, reg19);
        vaddps(reg08, reg08, reg20);
    }
}
```

Source: OneDNN

Python program to generate ASM

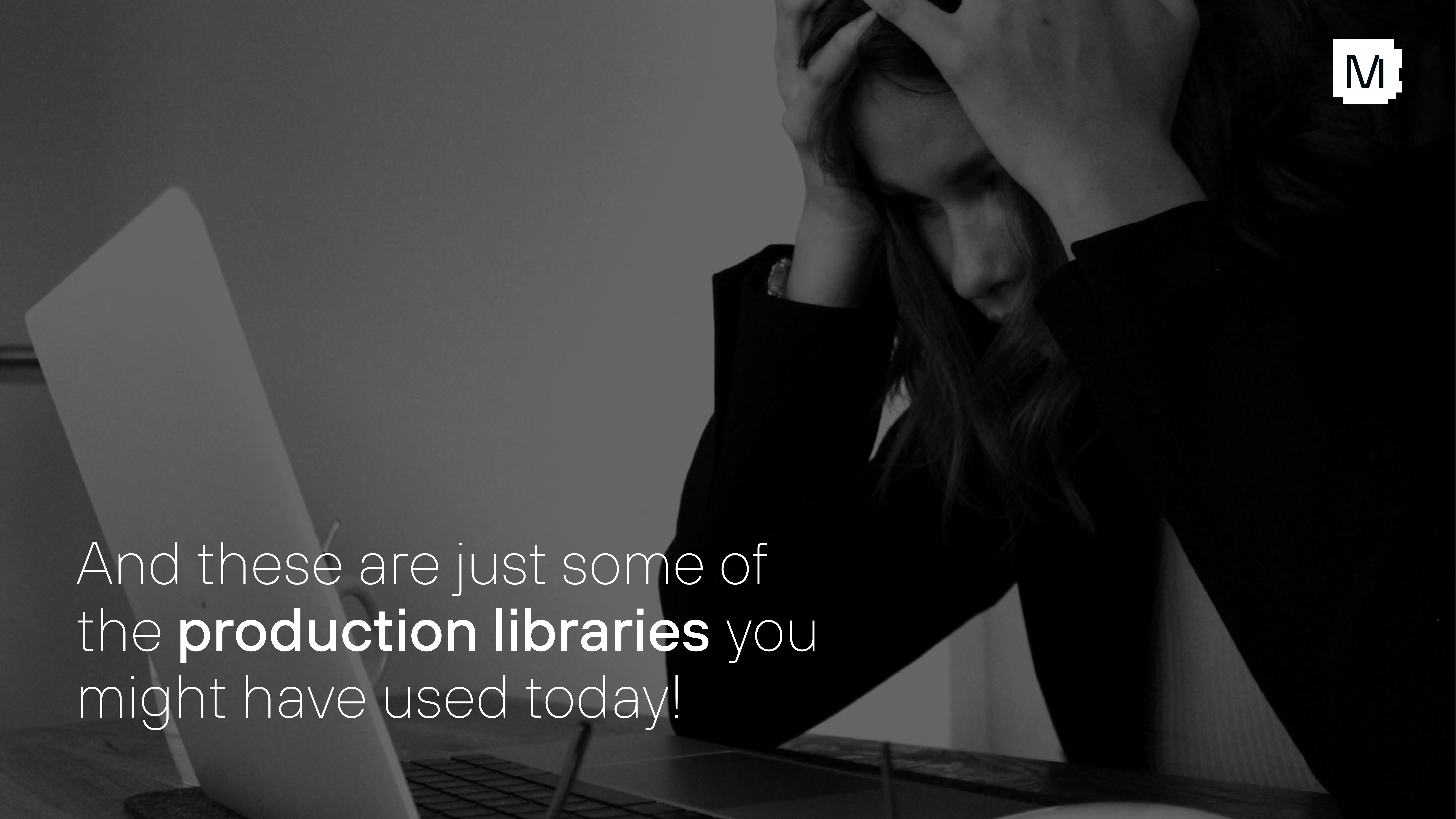
```python
for iui in range(0, innerUnroll):
    for idx1 in range(0, kernel["ThreadTile1"]):
        for idx0 in range(0, kernel["ThreadTile0"]):
            vars["idx0"] = idx0
            vars["idx1"] = idx1
            vars["a"] = idx0 if writer.tPB["tile01Idx"] else idx1
            vars["b"] = idx1 if writer.tPB["tile01Idx"] else idx0
            vars["iui"] = iui

            vars["cStr"] = "v[vgprValuC + {idx0} + {idx1}*{ThreadTile0}]".format_map(vars)
            vars["aStr"] = "v[vgprValuA_X{m}_I{iui} + {a}]".format_map(vars)
            vars["bStr"] = "v[vgprValuB_X{m}_I{iui} + {b}]".format_map(vars)

            if instruction == "v_fma_f32":
                kStr += "v_fma_f32 {cStr}, {aStr}, {bStr}, {cStr}{endLine}".format_map(vars)
            else:
                kStr += "{instruction} {cStr}, {aStr}, {bStr}{endLine}".format_map(vars)

            kStr += priority(writer, 1, "Raise priority while processing macs")
```

Source: Tensile

Python template to generate C++

```
const __m128i vsign_mask =
    _mm_load_si128((const __m128i*)params->${PARAMS_STRUCT}.sign_mask);
const __m256 vsat_cutoff = _mm256_load_ps(params->${PARAMS_STRUCT}.sat_cutoff);
const __m256 vlog2e = _mm256_load_ps(params->${PARAMS_STRUCT}.log2e);
const __m256 vmagic_bias = _mm256_load_ps(params->${PARAMS_STRUCT}.magic_bias);
const __m256 vminus_ln2 = _mm256_load_ps(params->${PARAMS_STRUCT}.minus_ln2);
$for i in reversed(range(2, P + 1))
    : const __m256 vc${i} = _mm256_load_ps(params->${PARAMS_STRUCT}.c${i});
$if P != H + 1 : const __m256 vminus_one =
    _mm256_load_ps(params->${PARAMS_STRUCT}.minus_one);
const __m256 vtwo = _mm256_load_ps(params->${PARAMS_STRUCT}.two);
$if P == H + 1 : const __m256 vminus_one =
    _mm256_load_ps(params->${PARAMS_STRUCT}.minus_one);
```

Source: XNNPack



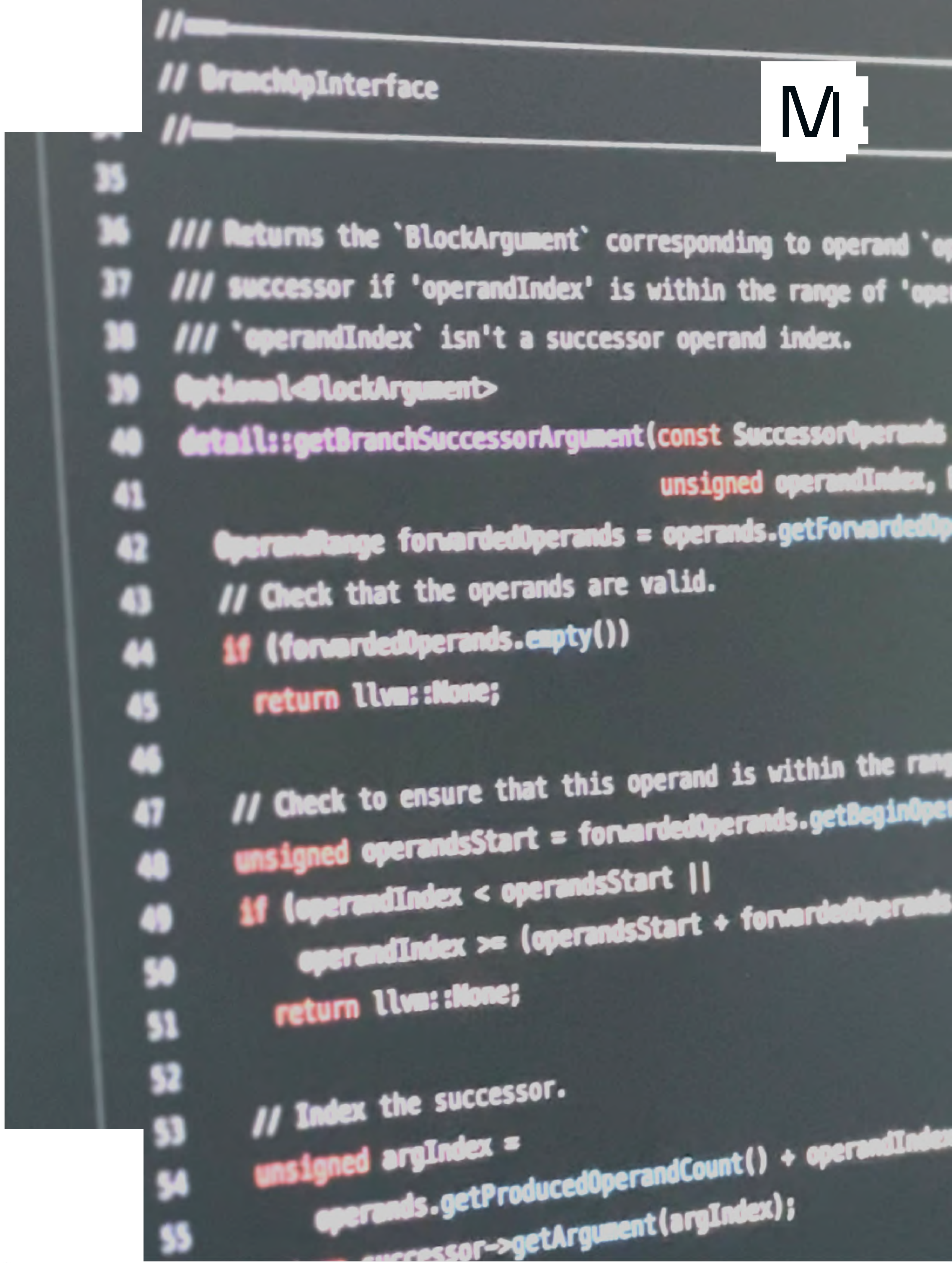
And these are just some of the **production libraries** you might have used today!

You lose on so much

Maintainability, debugging, tooling, ...

Hackability has suffered with binary library distributions

- Libraries contain the program semantics and hardware specifics
- Higher level compilers (e.g. graph compilers) cannot reason about them
- Users cannot extend them and hardware vendors cannot retarget them
- You end up with point-solutions (Conv + Activation+enum) of stamped popular patterns
- No consistent API, distribution story, …



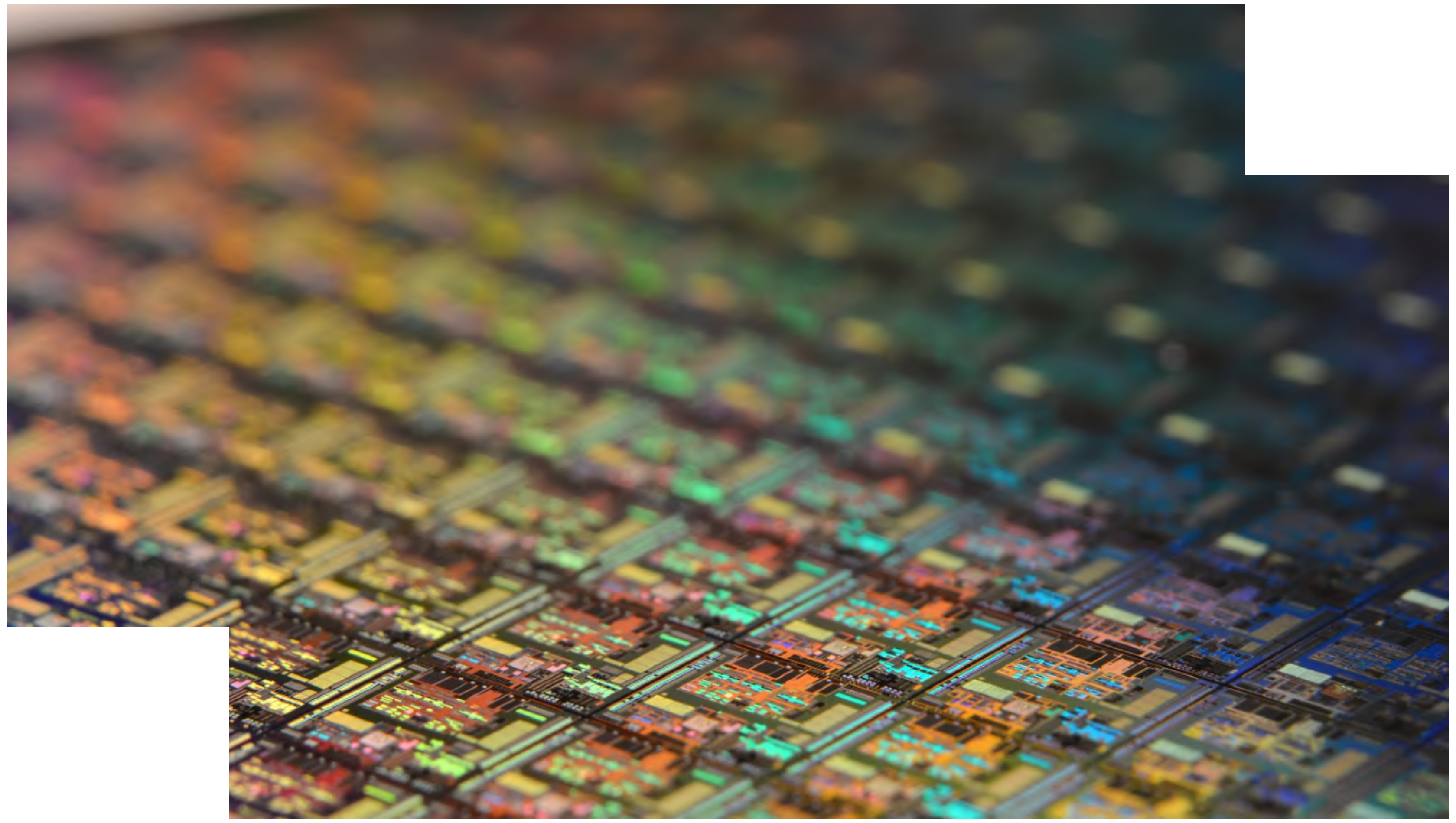
This is why we built

Mojo 🔥

Let's help the developer

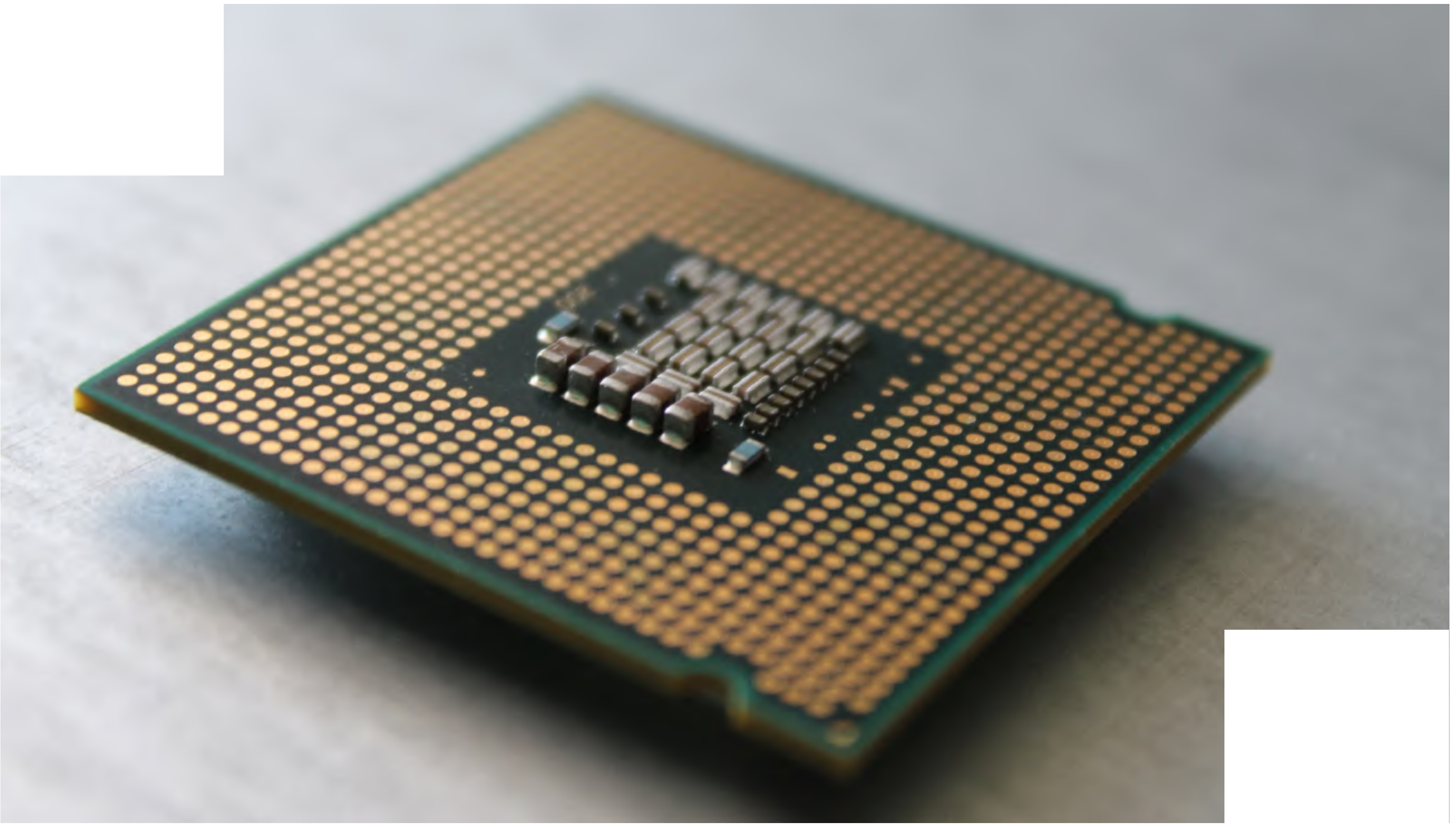
- Put optimizations into the library rather than the compiler
- Leverage humans for what they are good at and computers where they are good at
  - Computers are great for searching – can be brute force or intelligent
  - Search for right parameters or combination of algorithms
  - Search can be distributed across N machines
- Give them the tools to be productive

Let's help the developer



SIMD is a core type

- Parametric on width and type
- Scalars are SIMD type with a width of 1
- All math functions work on SIMD elements



Parallelism and asynchrony

- Built in from the beginning making it more usable and natively accessible

Power to the developer

The full power of the silicon is available in Mojo:

- Access to all hardware intrinsics in LLVM and MLIR
- Ability to write inline assembly
- Target any LLVM/MLIR backend

Mojo is a general purpose programming language

- Not limited in any way to “just AI”

Implementing compiler infrastructure in Mojo🔥 as libraries

Mojo uses MLIR core, but few standard dialects

We use LLVM and index dialect:

- do not use arith, vector, affine, MemRef, Linalg, etc

Several reasons:

- They are not always production quality
- They do not always have full coverage
- These often have complex interdependencies
- Lowering is not always target hardware aware

Functionality is implemented in Mojo code as libraries

Vector reduction in Mojo

```
struct SIMD[type: DType, width: Int]:
    ...
    fn reduce_max(self) -> SIMD[type, 1]:
        @parameter
        if size == 1:
            return self[0]
        elif is_x86():
            ...
```

Vector reduction in Mojo

```
...
elif is_x86():
    fn reduce[type: DType, width: Int](val: SIMD[type, width]) -> SIMD[type, 1]:
        @parameter
        if size == 1:
            return val[0]
        elif size == 2:
            return max(val[0], val[1])

        alias half_width = width // 2
        let lhs = val.slice[half_width](0)
        let rhs = val.slice[half_width](half_width)
        return max(lhs.reduce_max(), rhs.reduce_max())

    return reduce(self)
elif type.is_floating_point():
  ...
```

Vector reduction in Mojo

```
...
elif is_x86():
    ...
elif type.is_floating_point():
    return llvm_intrinsic["llvm.vector.reduce.fmax"](self)
elif type.is_unsigned():
    return llvm_intrinsic["llvm.vector.reduce.umax"](self)
else:
    return llvm_intrinsic["llvm.vector.reduce.smax"](self)
```

Compare that to …

```cpp
/// Conversion pattern for all vector reductions.
Nicolas Vasilache, 2 weeks ago | 5 authors (Siva Chandra Reddy and others)
class VectorReductionOpConversion
    : public ConvertOpToLLVMPattern<vector::ReductionOp> {
public:
  explicit VectorReductionOpConversion(const LLVMTypeConverter &typeConv,
                                       bool reassociateFPRed)
      : ConvertOpToLLVMPattern<vector::ReductionOp>(typeConv),
        reassociateFPReductions(reassociateFPRed) {}

  LogicalResult
  matchAndRewrite(vector::ReductionOp reductionOp, OpAdaptor adaptor,
                  ConversionPatternRewriter &rewriter) const override {
    auto kind = reductionOp.getKind();
    Type eltType = reductionOp.getDest().getType();
    Type llvmType = typeConverter->convertType(eltType);
    Value operand = adaptor.getVector();
    Value acc = adaptor.getAcc();
    Location loc = reductionOp.getLoc();

    if (eltType.isIntOrIndex()) {
      // Integer reductions: add/mul/min/max/and/or/xor.
      Value result;
      switch (kind) {
      case vector::CombiningKind::ADD:
        result =
            createIntegerReductionArithmeticOpLowering<LLVM::vector_reduce_add,
                                                       LLVM::AddOp>(
                rewriter, loc, llvmType, operand, acc);
        break;
      case vector::CombiningKind::MUL:
        result =
            createIntegerReductionArithmeticOpLowering<LLVM::vector_reduce_mul,
                                                       LLVM::MulOp>(
                rewriter, loc, llvmType, operand, acc);
        break;
      case vector::CombiningKind::MINUI:
        result = createIntegerReductionComparisonOpLowering<
            LLVM::vector_reduce_umin>(rewriter, loc, llvmType, operand, acc,
                                      LLVM::ICmpPredicate::ule);
        break;
      case vector::CombiningKind::MINSI:
        result = createIntegerReductionComparisonOpLowering<
            LLVM::vector_reduce_smin>(rewriter, loc, llvmType, operand, acc,
                                      LLVM::ICmpPredicate::sle);
        break;
      case vector::CombiningKind::MAXUI:
        result = createIntegerReductionComparisonOpLowering<
            LLVM::vector_reduce_umax>(rewriter, loc, llvmType, operand, acc,
                                      LLVM::ICmpPredicate::uge);
        break;
```

```cpp
      case vector::CombiningKind::MAXSI:
        result = createIntegerReductionComparisonOpLowering<
            LLVM::vector_reduce_smax>(rewriter, loc, llvmType, operand, acc,
                                      LLVM::ICmpPredicate::sge);
        break;
      case vector::CombiningKind::AND:
        result =
            createIntegerReductionArithmeticOpLowering<LLVM::vector_reduce_and,
                                                       LLVM::AndOp>(
                rewriter, loc, llvmType, operand, acc);
        break;
      case vector::CombiningKind::OR:
        result =
            createIntegerReductionArithmeticOpLowering<LLVM::vector_reduce_or,
                                                       LLVM::OrOp>(
                rewriter, loc, llvmType, operand, acc);
        break;
      case vector::CombiningKind::XOR:
        result =
            createIntegerReductionArithmeticOpLowering<LLVM::vector_reduce_xor,
                                                       LLVM::XOrOp>(
                rewriter, loc, llvmType, operand, acc);
        break;
      default:
        return failure();
      }
      rewriter.replaceOp(reductionOp, result);

      return success();
    }

    if (!isa<FloatType>(eltType))
      return failure();

    arith::FastMathFlagsAttr fMFAttr = reductionOp.getFastMathFlagsAttr();
    LLVM::FastmathFlagsAttr fmf = LLVM::FastmathFlagsAttr::get(
        reductionOp.getContext(),
        convertArithFastMathFlagsToLLVM(fMFAttr.getValue()));
    fmf = LLVM::FastmathFlagsAttr::get(
        reductionOp.getContext(),
        fmf.getValue() | (reassociateFPReductions ? LLVM::FastmathFlags::reass
                                                  : LLVM::FastmathFlags::none)

    // Floating-point reductions: add/mul/min/max
    Value result;
    if (kind == vector::CombiningKind::ADD) {    repeated branch body in condi
      result = lowerReductionWithStartValue<LLVM::vector_reduce_fadd,
```

```cpp
                                            ReductionNeutralZero>(
          rewriter, loc, llvmType, operand, acc, fmf);
    } else if (kind == vector::CombiningKind::MUL) {
      result = lowerReductionWithStartValue<LLVM::vector_reduce_fmul,
                                            ReductionNeutralFPOne>(
          rewriter, loc, llvmType, operand, acc, fmf);
    } else if (kind == vector::CombiningKind::MINIMUMF) {
      result =
          createFPReductionComparisonOpLowering<LLVM::vector_reduce_fminimum>(
              rewriter, loc, llvmType, operand, acc, fmf);
    } else if (kind == vector::CombiningKind::MAXIMUMF) {
      result =
          createFPReductionComparisonOpLowering<LLVM::vector_reduce_fmaximum>(
              rewriter, loc, llvmType, operand, acc, fmf);
    } else if (kind == vector::CombiningKind::MINF) {
      result = createFPReductionComparisonOpLowering<LLVM::vector_reduce_fmin>(
          rewriter, loc, llvmType, operand, acc, fmf);
    } else if (kind == vector::CombiningKind::MAXF) {
      result = createFPReductionComparisonOpLowering<LLVM::vector_reduce_fmax>(
          rewriter, loc, llvmType, operand, acc, fmf);
    } else
      return failure();

    rewriter.replaceOp(reductionOp, result);
    return success();
  }

private:
  const bool reassociateFPReductions;
};
```

Writing transforms as library functions

```
fn vectorize[simd_width: Int,
             func: fn[width: Int](Int) capturing -> None](size: Int):
    # Process a simd_width at a time.
    for i in range(0, size, simd_width):
        func[simd_width](i)

    # Handle left-over elements with scalars.
    for i in range(simd_width * (size // simd_width), size):
        func[1](i)
```

What does this mean to the developer?

Performance engineers don’t need to be compiler engineers

01

You do not have to know what a dialect is or use TableGen.

02

You can invent new optimizations that do not exist in the compiler.

03

You can develop point-solutions for important specific problems.

Mojo 🔥 Performance Results

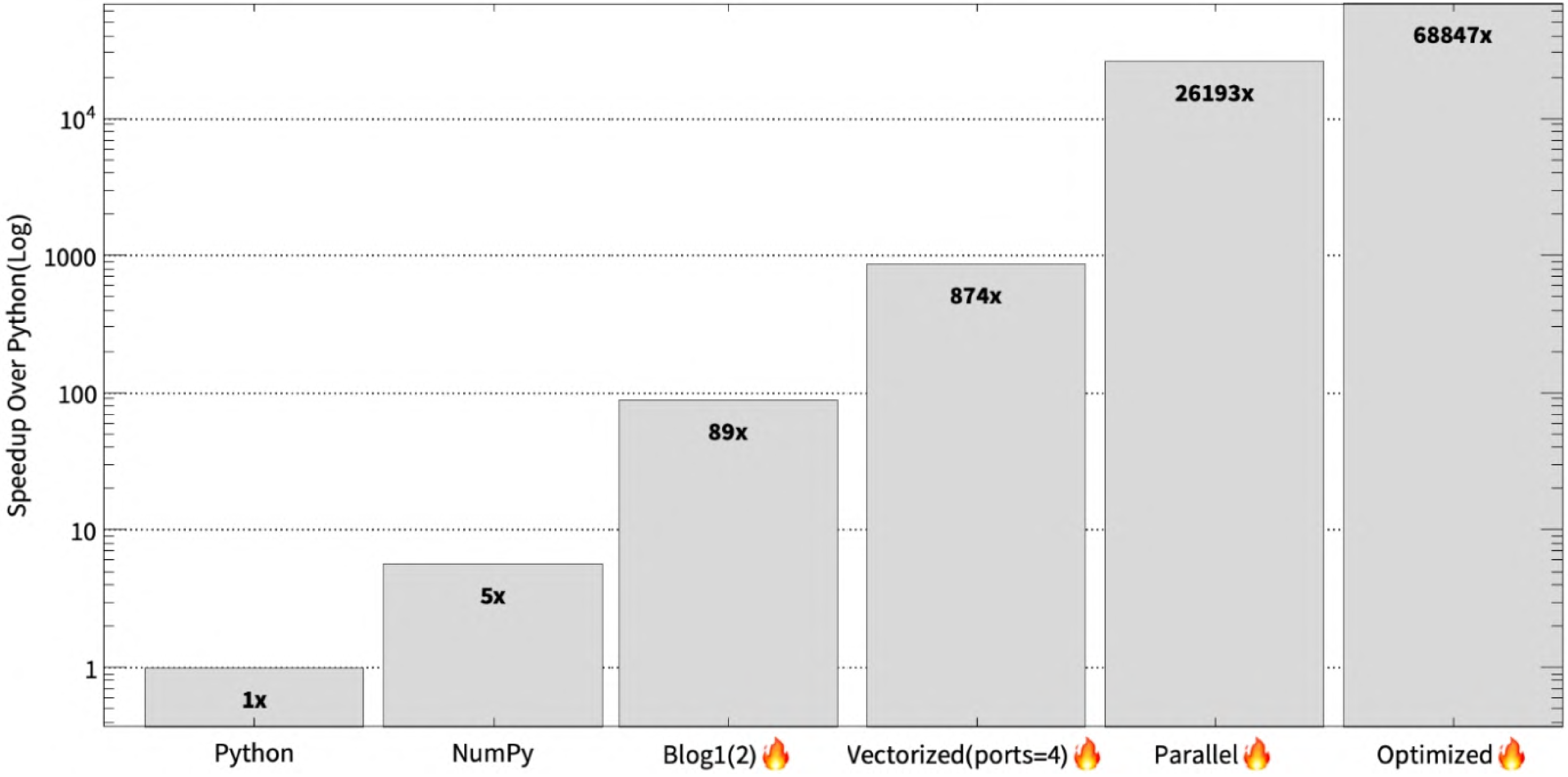
Mandlebrot

Mojo 🔥 is 68,000x times faster than Python 🐍

[Read our blog on this now!]

```
var in_set_mask: SIMD[DType.bool, simd_width] = True
for i in range(MAX_ITERS):
    if not in_set_mask.reduce_or():
        break
    in_set_mask = z.squared_norm() <= 4
    iters = in_set_mask.select(iters + 1, iters)
    z = z.squared_add(c)
return iters
```

Mandelbrot performance



Matrix Multiplication

Studied extensively since the 60s

- In 2023 there were 2k papers on GEMM

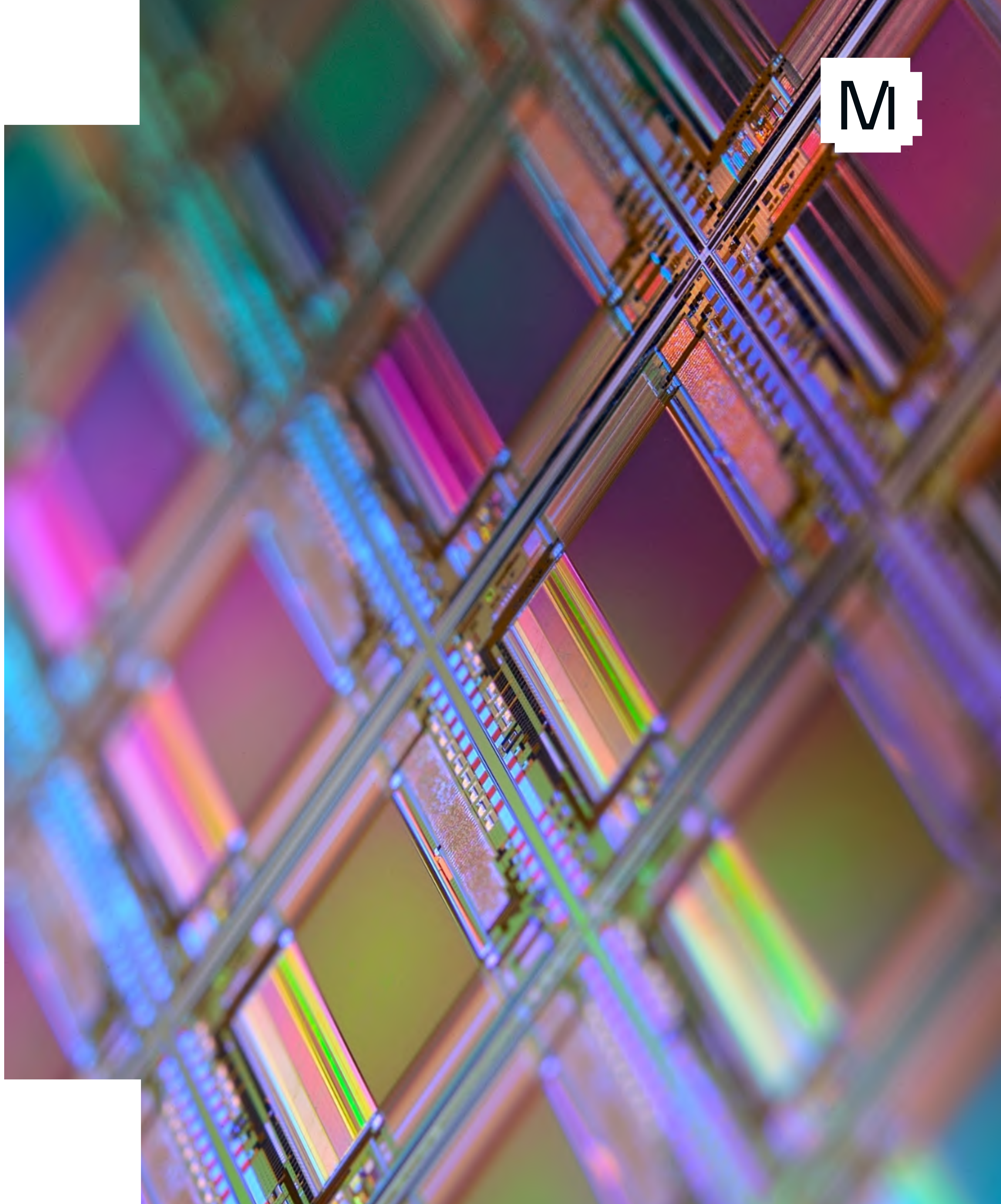
Optimal codegen is µarch dependent

- Size of L$
- Number of ports
- Types of instructions available

Core part of LAPACK and ML workloads

- Hardware companies are incentivized to optimize performance for benchmarks
- Part of core business for some companies

Libraries have been in development for decades



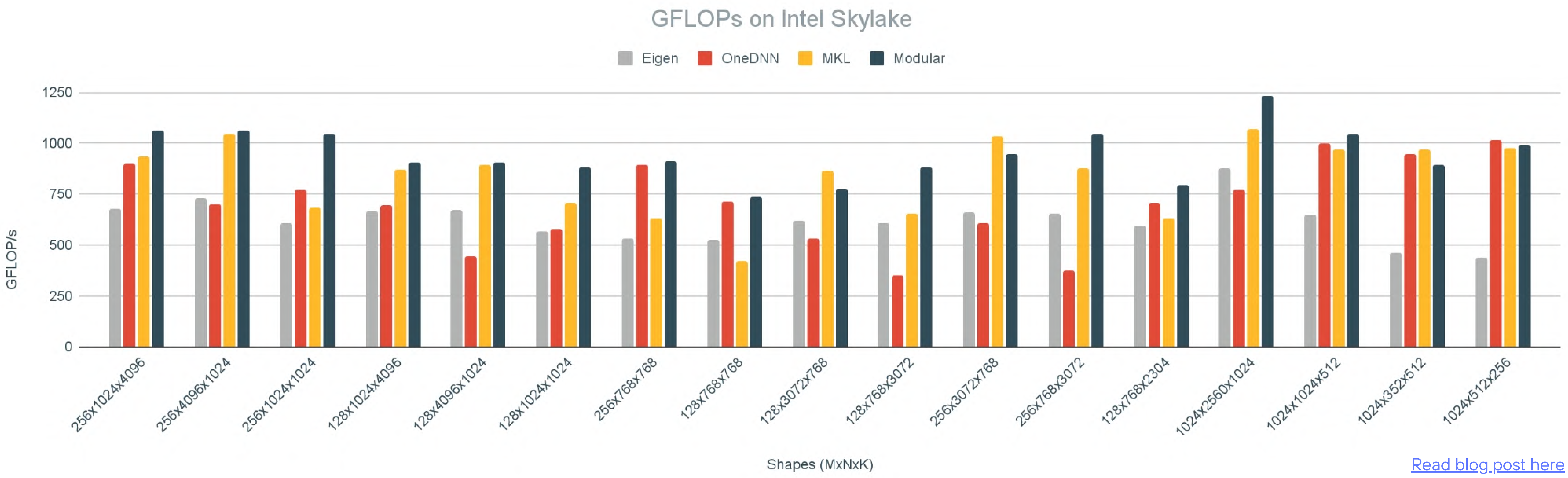
Goals for Matmul in Mojo

- Single source of truth
- Competes with SotA
- No assembly/C++/...
- Amenable to fusion
- Works on dynamic shapes, can also be specialized
- Works across all CPU architectures (VNNI, AVX512, NEON, AVX2, ...)
- Supports packing, different transpose modes, ...

... our core hypothesis from the beginning!

Matmul performance

1.46x faster than OneDNN on **Intel**

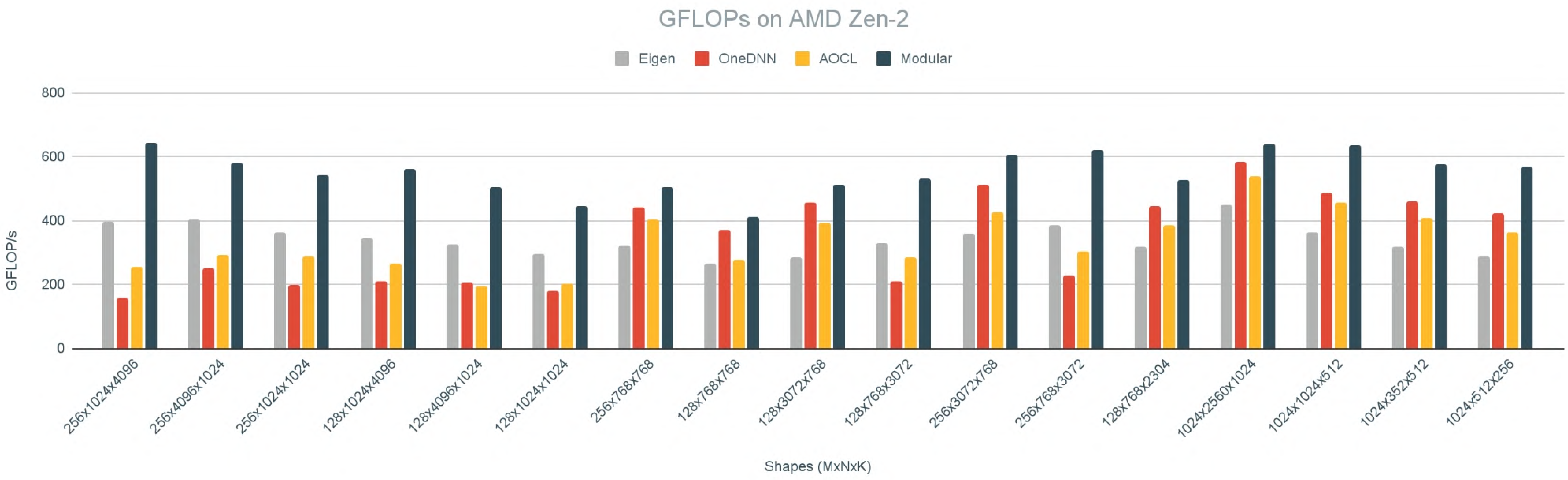


[Read blog post here]

Fully dynamic, no pre-packing, and no inlined assembly!

Matmul performance

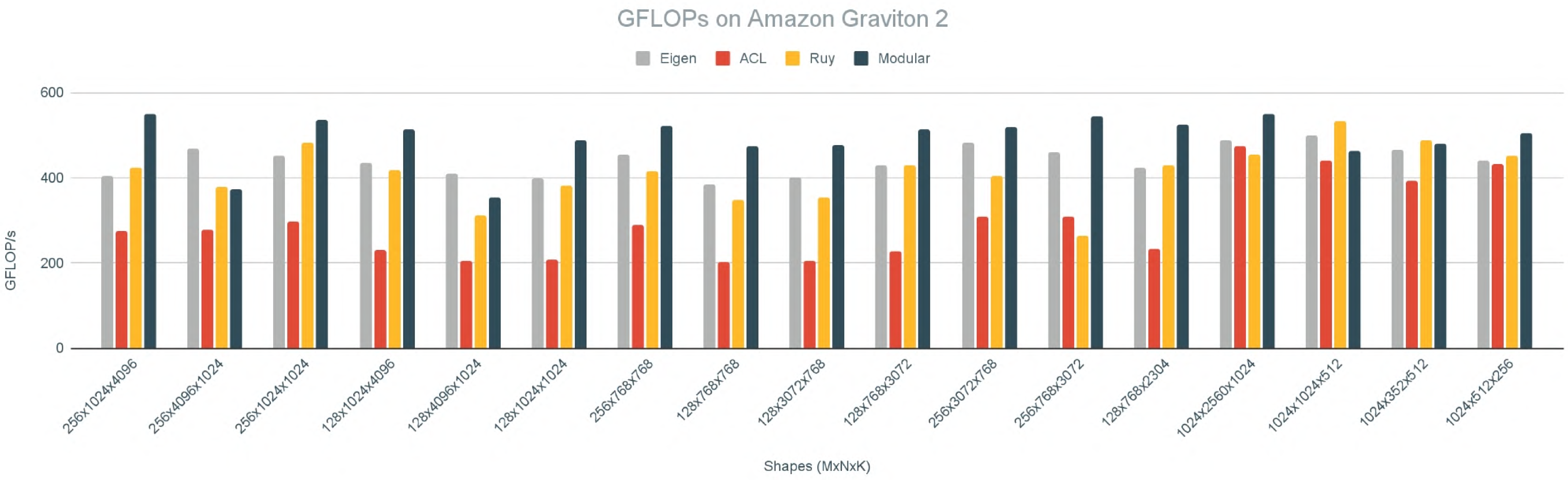
1.6x faster than SotA on **AMD**



Read blog post here

Matmul performance

1.2x faster than RUY on ARM



Read blog post here

Toy tiled Matmul implementation

```
fn matmul(C: Matrix, A: Matrix, B: Matrix):
    fn calc_row(m: Int):
        fn calc_tile[tile_x: Int, tile_y: Int](x: Int, y: Int):
            for k in range(y, y + tile_y):
                fn dot[nelts: Int](n: Int):
                   C.store[nelts](m,n+x,
                         C.load[nelts](m,n+x) + A[m,k] * B.load[nelts](k,n+x))

                vectorize_unroll[nelts, tile_x // nelts, dot](tile_x)

        # Let Mojo pick the best tile size!
        alias tile_size = autotune(1, 2, 4, 8, 16, 32)
        tile[calc_tile, nelts * tile_size, tile_size](A.cols, C.cols)

    parallelize[calc_row](C.rows, C.rows)
```

Hypothesis validated

We can build high performance portable libraries

Less suffering

With Mojo you get performance and generality in a production language


Mojo 🔥 Roadmap

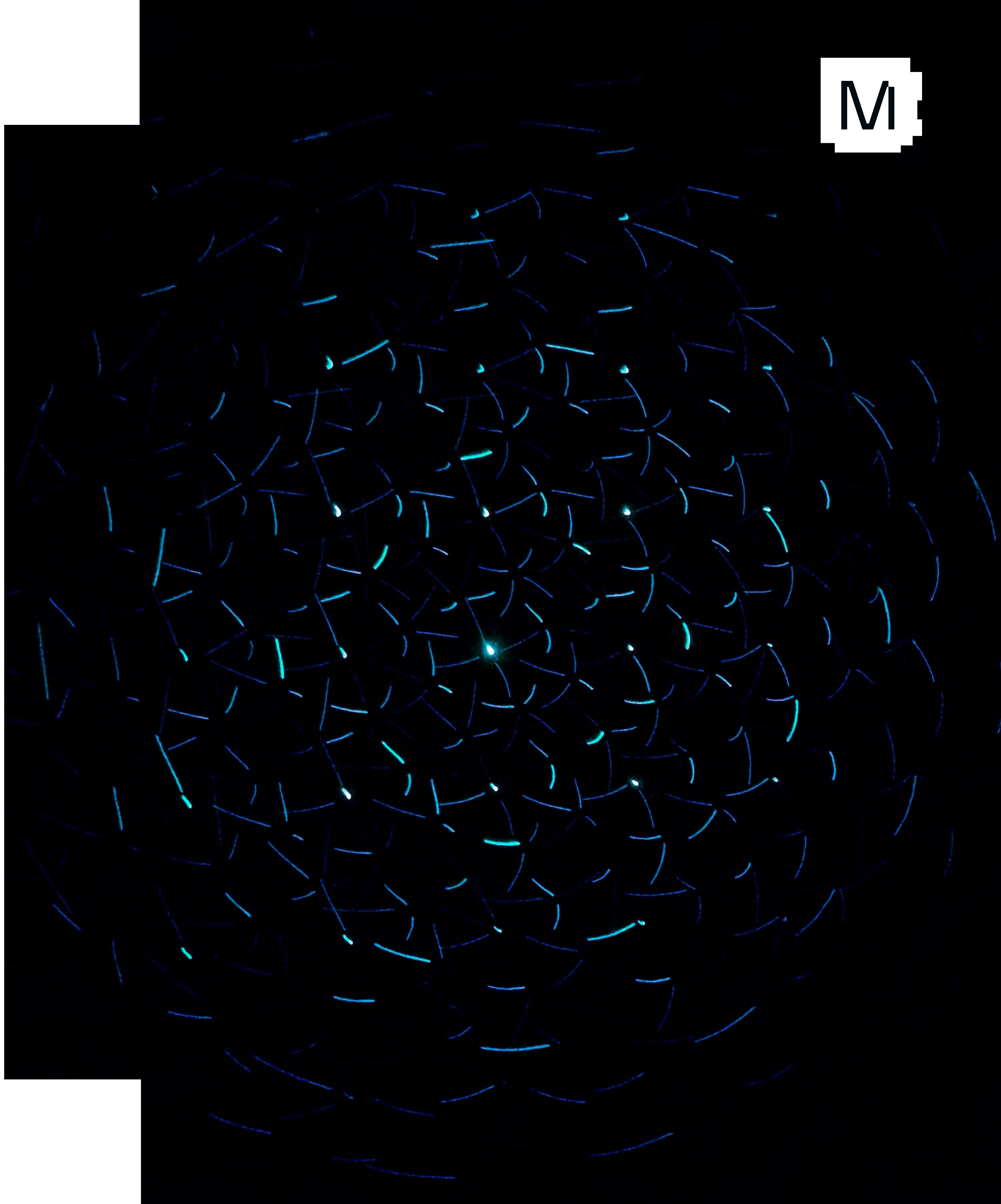
Mojo 🔥 Development Roadmap

Mojo is *useful* but still not done:

- Many features in development
- Prioritizing quality over time to market

New releases roll out every few weeks

[Read our Public Roadmap!]



Open Source?

Many contributions to LLVM upstream:

- MLIR Bytecode serialization
- MLIR Resources
- MLIR debug info support
- MLIR index dialect
- MLIR interpreter (soon?)

We will start opening Mojo 🔥 itself later this year!

Read more details here

open source initiative®

Mojo 🔥 +
Modular AI Engine = ❤️‍🔥

Mojo unlocks programmability for any one device:

- … and communities of developers

AI Engine unlocks heterogeneous computers:

- Distributed, asynchronous, accelerated
- Rapidly evolving architectures

More technical details at:
Workshop on ML for Systems at NeurIPS

Modular

Mojo

Download Now

https://www.modular.com/mojo