Modular

MAX

Unlocking developer productivity across CPU+GPU with MAX 🔥

"Generative AI is poised to unleash the next wave of productivity."

McKinsey 2023 AI Economic Report

We are pausing new ChatGPT Plus sign-ups for a bit :(

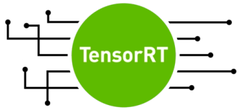
the surge in usage post devday **has exceeded our capacity** and we want to make sure everyone has a great experience.

Sam Altman
@sama

7:10PM • Nov 14, 2023

GenAI is nearly impossible to scale, even for the world's largest companies.

Caffe



OpenVINO™

TensorRT-LLM

text-generation-inference

vLLM

DeepSpeed

ggml

gemma.cpp

Model Training

2015, 2016, 2017

Model Inference

2018, 2019, 2020

GenAI & LLMs

2022-Present

Explosion of **model-specific** inference frameworks

What happened to reusable AI Infrastructure?

Traditional AI was simple...

Just a graph	→	Easy export and manipulate
Fixed operator sets	→	Big hard coded kernel libraries
Simple numerics	→	Few worried about quantization
Simple batching	→	Hard coded serving

GenAI **broke** those assumptions

Not just a graph	→	Ensembles, MoE's, chained models
Customized operators	→	Attention comes in many variants
Complex numerics	→	New datatypes every year
Complex orchestration	→	Fully custom serving solutions

Classic AI infrastructure

❌ **Old technology**, hill climbed

❌ **Fragmentation** and **complexity**

❌ Explosion of **novel hardware**

❌ **Hackability died** along the way

Complexity is **hurting us all**

We need **fewer things**,
that **work better**!

MAX

MODULAR ACCELERATED EXECUTION

MAX Platform

Mojo 🔥

MAX Engine

MAX Serving ⚡

Mojo 🔥

One language for
all AI programming

Mojo 🔥 at a glance

Pythonic **system programming** language

- *Faster* than Rust; runs on GPUs
- Full toolchain + VSCode LSP support

Freely available on Linux, Mac and Windows

- Progressively open sourcing it as **Apache 2**

A growing and vibrant community

- 175K+ users overall, 21K+ users on Discord

Well funded, long term perspective

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


Python Dev +
Ecosystem
Compatibility
Next generation
PL + compiler
design

Forget your preconceptions about Python! 🐍

Languages are hard!

We've done this before:

OpenCL™ Clang Swift

We've learned you need:

- ✔ Consistent vision
- ✔ Long term commitment
- ✔ Ability to attract specialized talent
- ✔ Big target market of developers
- ✔ Vibrant community

Our mission: **advance CPU+GPU AI innovation**

- We'll do "hard things" for a better result

Easy to use!

One language that scales:

- Full Python ecosystem integration
- High performance CPU + GPU logic

Tools that can scale:

- REPL, notebooks, compiler, debugger, etc

Lift developers incrementally:

- Easy to get started and learn progressively
- Dynamic and fun to work with
- Bring new people into GPU programming!

Mandelbrot in Mojo with Python plots

Not only is Mojo great for writing high-performance code, but it also allows us to leverage the huge Python ecosystem of libraries and tools. With seamless Python interoperability, Mojo can use Python for what it's good at, especially GUIs, without sacrificing performance in critical code. Let's take the classic Mandelbrot set algorithm and implement it in Mojo.

This tutorial shows two aspects of Mojo. First, it shows that Mojo can be used to develop fast programs for irregular applications. It also shows how we can leverage Python for visualizing the results.

```
#|code-fold: true ···
```

```
def show_plot(tensor: Tensor[int_type]): ···
```

GPU

```
@always_inline
fn mandelbrot_kernel[
    simd_width: Int
](c: ComplexSIMD[float_type, simd_width]) -> SIMD[int_type, simd_width]:
    var z = c
    for i in range(MAX_ITERS):
        z = z * z + c
        if z.squared_norm() > 4:
            return i
    return MAX_ITERS
```

[3] mojo

CPU

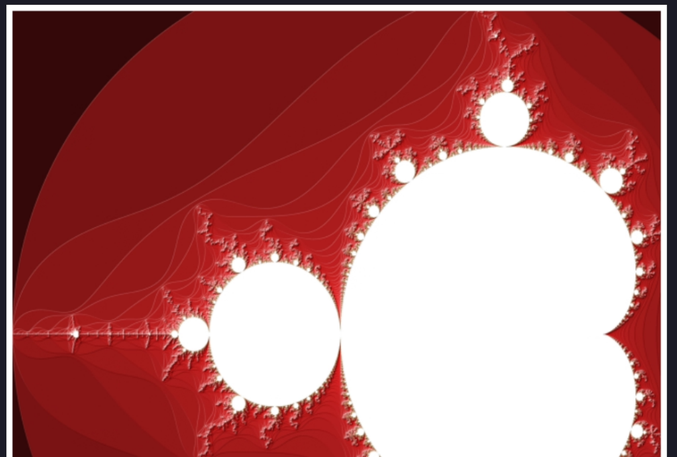
```
fn mandelbrot(out: NDBuffer[int_type, 2, DimList(height, width)]): ···
```

```
alias BLOCK_SIZE = 128 ···
```

```
show_plot(run_mandelbrot())
```

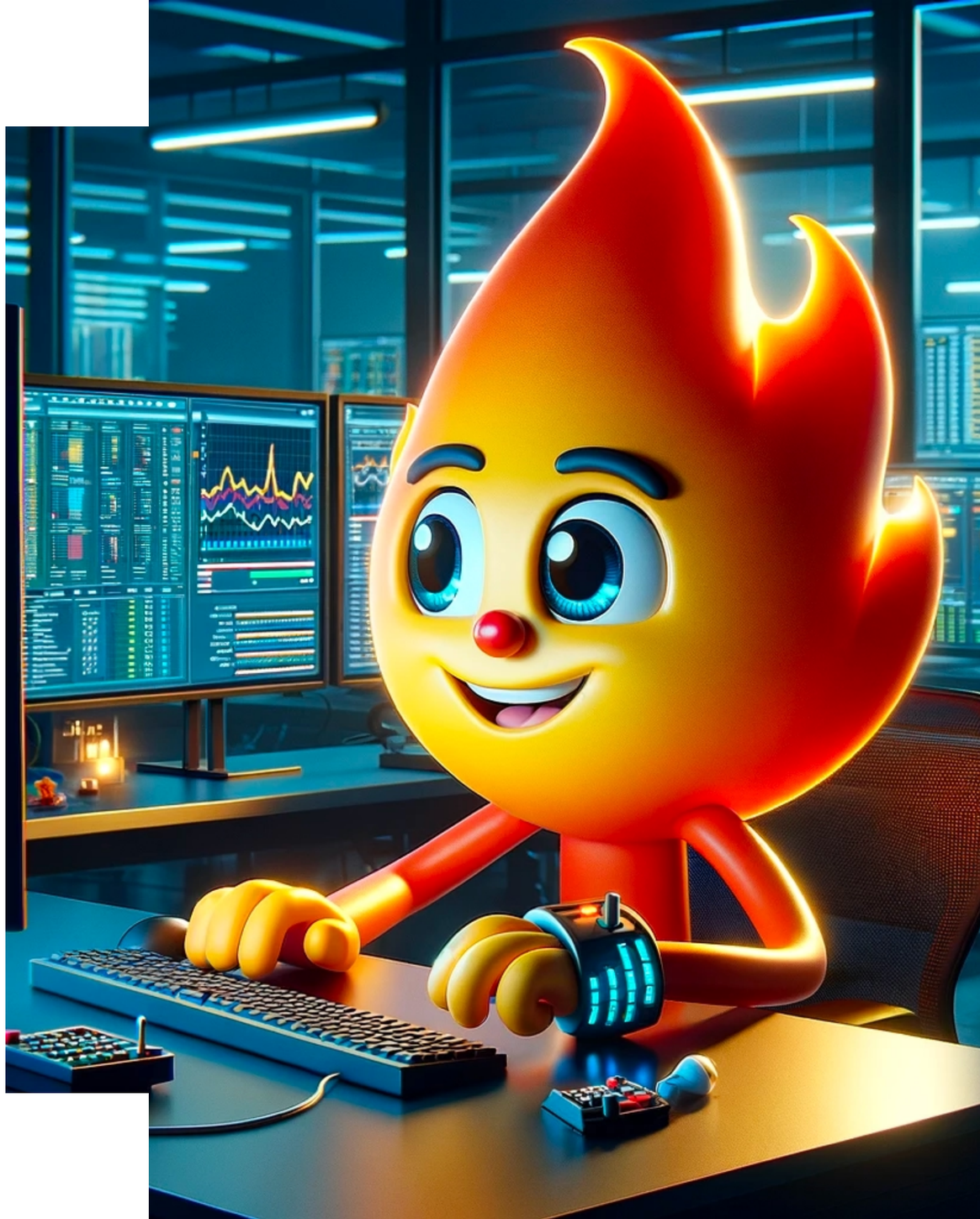
[7] mojo

```
Computation took: 155.72759459016393
```



Python

Superpowers for Library developers



Most languages are built on hard coded magic

Consider C++:

- `double` is hard coded in language
- `std::complex` is defined in the library

Consider Python:

- Core types are implemented in C, not Python
- Python code is for high level and glue

Hardware pace was slow in the 1990's

Mojo is a library-first language 📚

AI Software/Hardware innovation moves incredibly fast!

- Hello FP4!

Much "language" design is built in libraries!

- Int, Float, SIMD, String, Array, all written in Mojo itself

Mojo's purpose is to enable expressive libraries

- Talk to low level MLIR/LLVM compiler technologies. 🤓
- Talk to all the weird hardware 🔨

What does this mean for Mojo devs?

You can innovate without being a compiler engineer!

01

You do **not** have to know **compilers like LLVM/MLIR**

... but if you do, you can wrap their power into nice libraries for your clients

02

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

03

You can develop point-solutions for important specific problems

Compile Time Metaprogramming

Mojo🔥 needs ...
... what Python 🐍 has

Powerful metaprogramming:

- Decorators
- Metaclasses
- Reflection

But ... Runtime metaprogramming is slow
... it will never run on the GPU!

💡 Let's do it at compile time! 💡

Meta-programming with Mojo🔥

01

Mojo's metaprogramming language **is just Mojo** 🔥

02

Interpreter evaluates code... **at compile time**

03

Almost **any user-defined type** can be directly used

Simple compile time parameters

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

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

```
# "alias" declarations => comptime calculations
alias Float32 = SIMD[DType.f32, 1]
alias MAGIC_NUMBER = 4+5
```

~= C++ templates and `#define` and `typedef` ...

... but more powerful and easier to use

Function can be called at either compile or run time

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

List with heap allocation

List computed at compile-time...

```
def comptime_vector():
    alias lst = fill(15, 20)
    for e in lst:
      print(e)
```

used at runtime!

Mojo 🔥

High Performance across CPU+GPU

Built for low-level performance

- Zero cost abstractions
- Thin runtime
- Native support for SIMD
- Builtin optimizations like unrolling

```
for k in range(NR):
    b_next_tile = b_row.tile[1, NR](0, k + 4)
    @unroll
    for n in range(0, NR, simd_size):
        b_next_tile.prefetch(0, n)

    b_tile = b_row.tile[1, NR](0, k)

    @unroll
    for m in range(MR):
        av = a_tile[m, k]

        c_cache[m, 0] += av * b_tile.load[NR](0, 0)
```

Higher level libraries can abstract hardware complexity

- Packaged in Mojo standard library — ready to use
- Implemented using basic constructs
- You can choose the level of abstraction that suits the problem you are solving

```
fn swish(inout C: Matrix, A: Matrix):

    fn do_swish[width:Int](coords: StaticIntTuple):
        var vec = A.load[width=width][coords]
        C[coords] = vec / (1 + exp(-vec))

    elementwise[do_swish,
                width=simdwidthof[C.type],
                target="cuda"]((M,N))
```

What if a HW instruction is not supported yet?

```
fn shuffle_down(mask: Int, val: Scalar, offset: Int) -> Scalar:
  """Copies values from other lanes in the warp.
  Exchange a value between threads within a warp by copying from
  a thread with higher lane id relative to the caller without the
  use of shared memory.
  """
  @parameter
  if type.is_float32():
      return llvm_intrinsic["llvm.nvvm.shfl.sync.down.f32"](
          mask, val, offset, _WIDTH_MASK
      )
  elif ...:
```


What about
Matrix Multiplication?

Matrix Multiplication

Studied extensively since the 1960s

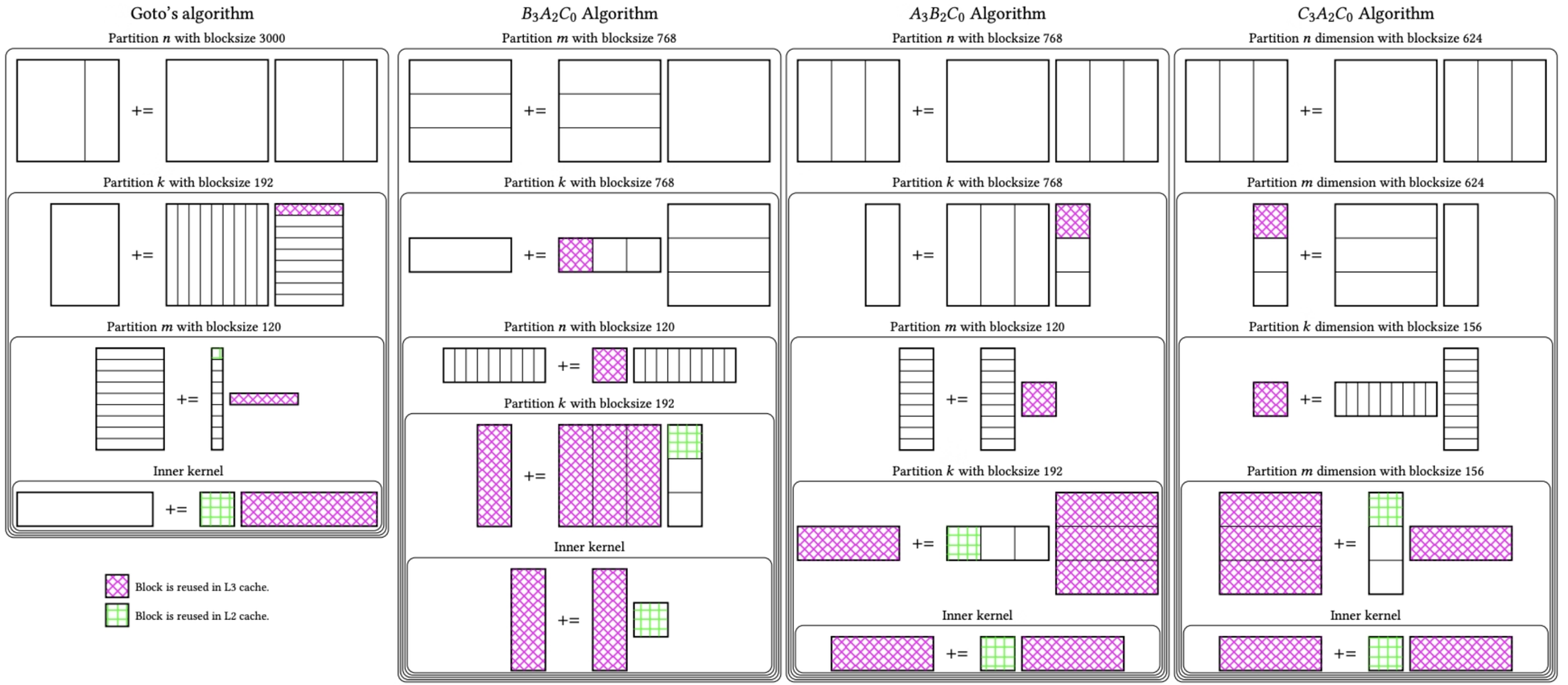
- In 2023 there were 2,000 papers on GEMM

Optimal codegen is μarch dependent

- Size of shared memory, number of SMs
- Availability of accelerators like tensor cores

Hand written libraries are considered SoTA

Optimizing Matrix Multiplication is Hard



Unifying CPU targets and GPUs might sound crazy!

https://arxiv.org/abs/1904.05717

Unlocking Matmul in Mojo

Our approach:

- Build abstractions in Mojo
- Compose into high-performance libraries

Our goal:

- Optimize E2E model performance
- Reduce implementation complexity
- Unlock research creativity
- Demonstrate hardware abstraction

Generalize to other problems!

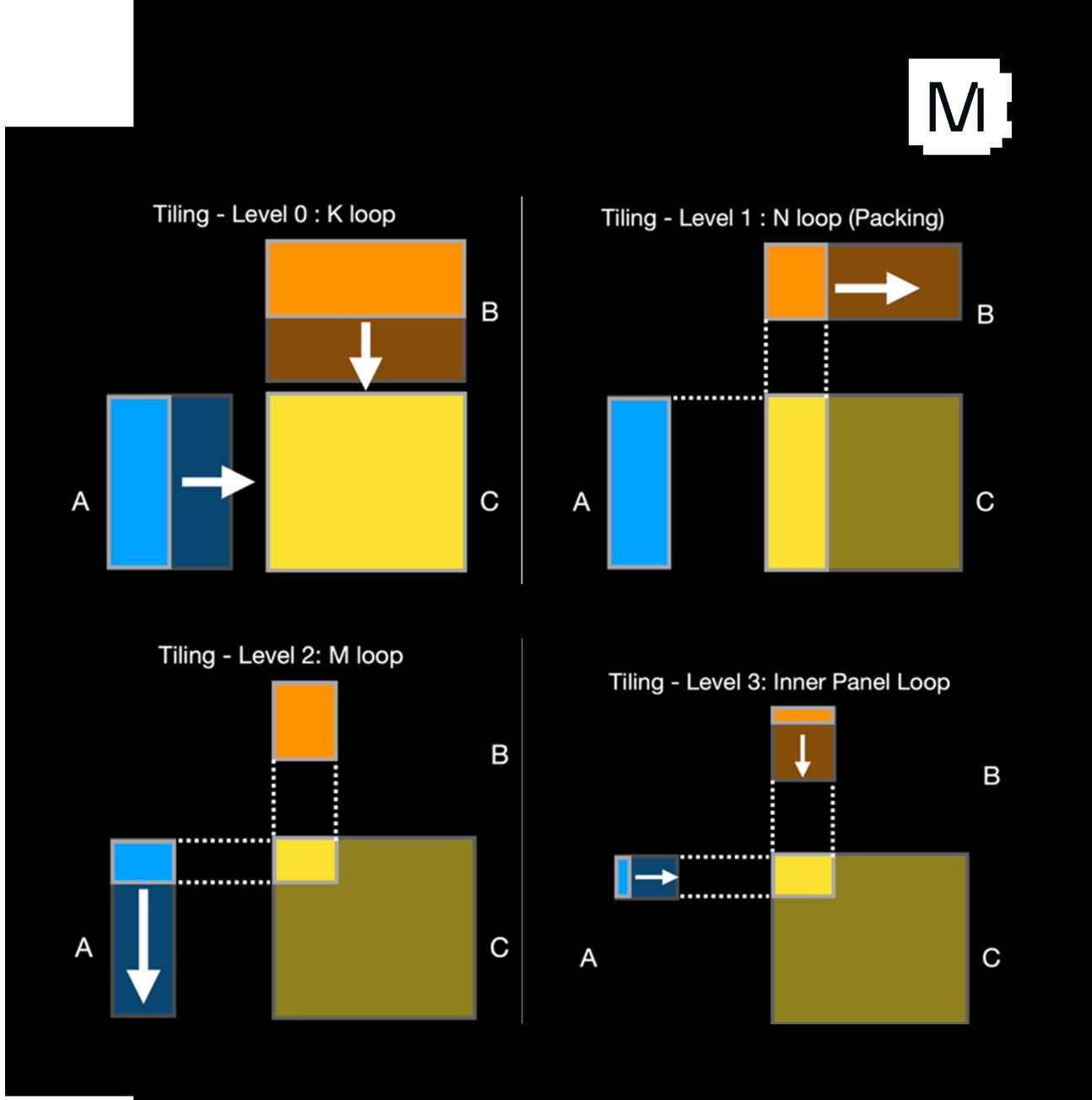
MatMul recursive decomposition

Self-similar structure:

- Blocking used at every level

Details make it hard:

- Indexing details
- Software pipelining
- Specialize hardware like TensorCores
- DType specific hacks



Tile Layouts

Encode the logical to physical indexing, simplifying complex indexing like swizzling.

E.g TF32 A100 register mapping can be expressed in Mojo by:

```
alias layout_a_mma = Layout(((4, 8), (2, 2)),((16, 1), (8, 64)))
alias layout_b_mma = Layout(((4, 8), 2), ((8, 1), 32))
alias layout_c_mma = Layout(((4, 8), (2, 2)),((32, 1), (16, 8)))
```

Row\Col	0	1	2	3	4	5	6	7
0	T0:a0	T1:a0	T2:a0	T3:a0	T0:a2	T1:a2	T2:a2	T3:a2
1	T4:a0	T5:a0	T6:a0	T7:a0	T4:a2	T5:a2	T6:a2	T7:a2
2								
..								
7	T28:a0	T29:a0	T30:a0	T31:a0	T28:a2	T29:a2	T30:a2	T31:a2
8	T0:a1	T1:a1	T2:a1	T3:a1	T0:a3	T1:a3	T2:a3	T3:a3
9	T4:a1	T5:a1	T6:a1	T7:a1	T4:a3	T5:a3	T6:a3	T7:a3
10								
..								
15	T28:a1	T29:a1	T30:a1	T31:a1	T28:a3	T29:a3	T30:a3	T31:a3

TF32 A100 A MMA fragment register mapping

Building this in Mojo🔥

Can be 0D (Scalar), 1D (Vector), 2D (Tensor fragment), ... ND

Placement can be at the warp-level, block-level, cluster-level, or device.

Enables warp-centric, block-centric, and device-centric programming.

```
struct Tile[
    dtype: DType, layout: Layout, *,
     placement: Placement]:

    fn tile[M: Int, N: Int,  *,
        layout: Layout =
         Self._compute_tile_layout[M, N](),
    ](self, m: Int, n: Int) -> Tile[
        dtype, layout[0], placement=placement
    ]:
    alias sm, sn = layout[1].stride
    var offset = m * sm + n * sn
    return Tile[
      dtype, layout[0], placement=placement
    ](self.ptr + offset)
```

Async Native

Async/await works in GPU kernels:

- Exposes SM asynchrony

User expreses data dependencies

Compiler can move code around

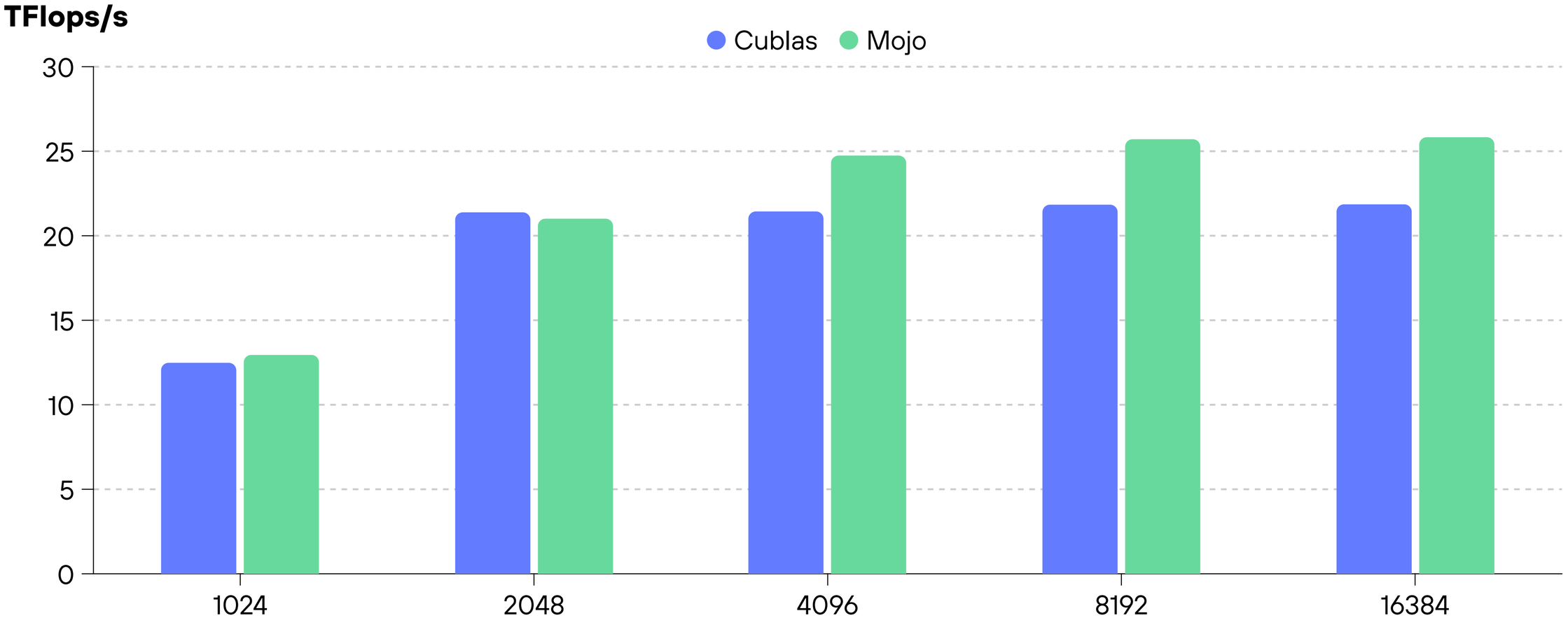
```
# Call to async function
var smem = input.copy[SHARED]()
. . .
other_operations()
. . .
use(await smem)
```

High Performance MatMul

- Built on top of Mojo and its libraries
- Compact, easy to maintain
- Parametrizable and extensible
- And ...

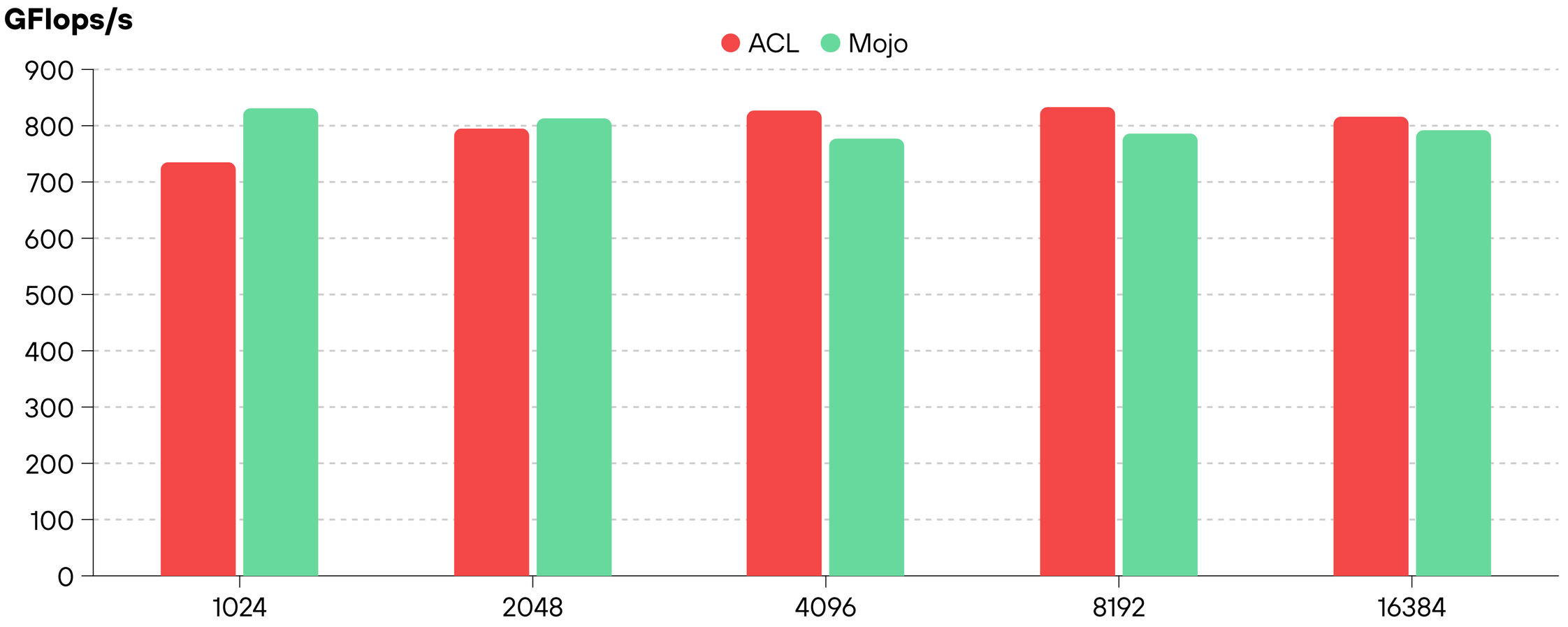
```
var a_frag = a[aRow, 0, shape=(16,16), layout=a_layout]
var b_frag = b[0, bCol, shape=(16,16), layout=b_layout]
var next_a_frag = Tile[shape=(16, 16), layout=a_layout]()
var next_b_frag = Tile[shape=(16, 16), layout=b_layout]()
var acc_frag = Tile[shape=(16, 16), layout=c_layout](0)
for i in range(16, K, 16):
   next_a_frag = a[i, aCol, shape=(16,16), layout=a_layout]
   next_b_frag = b[bRow, i, shape=(16,16), layout=b_layout]
   acc_frag += (await a_frag) @ (await b_frag)
   swap(a_frag, next_a_frag)
   swap(b_frag, next_b_frag)
acc_frag += (await a_frag) @ (await b_frag)
c.store(cRow, cCol, acc_frag, layout=c_layout)
```

Matches SoTA Performance



FP32 Gemm seqlen x seqlen x 128 (MxNxK) on RTX 4070 Ti

CPU SoTA Performance



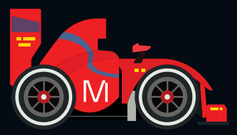
FP32 Gemm seqlen x seqlen x 128 (MxNxK), 16 Thread on Graviton3

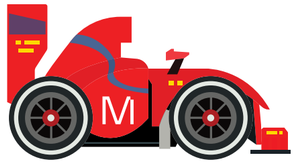
Tiles are not just for Matmul

```
def sum_reduction(buf: Tensor, global_atomic: Atomic):
  smem = stack_allocation[SHARED](blockDim.x)
  tile = buf.load_tile[WARP](tid)
  sum = tile.reduce(add)
  if not laneid:
    smem.store[SHARED](sum)
  if not threadIdx.x:
    result += await smem.reduce(add)
```

MAX

MAX Engine



MAX Engine  Graphs

Graph + Kernel compiler:

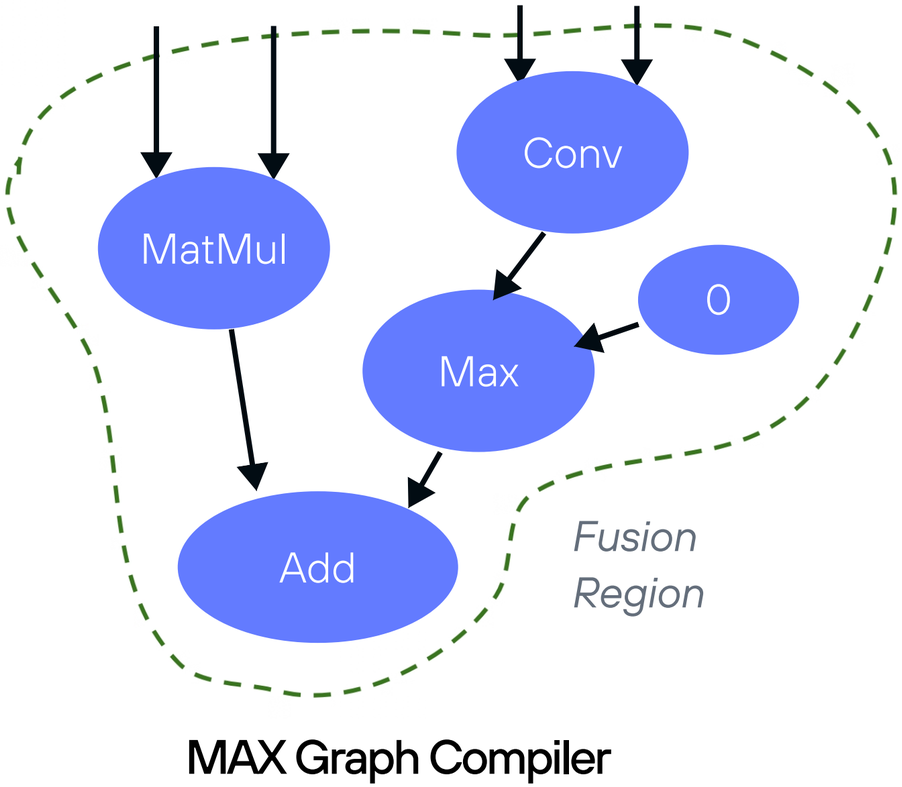
- Automatic kernel fusion
- Memory planning
- Multi-architecture support

State of the art:

- Full dynamic shapes, quantization, etc
- Surpasses HW vendor AI performance

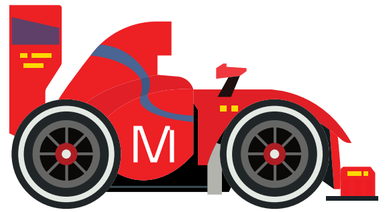
AI Framework support:

- MAX Native, PyTorch, ONNX, TensorFlow



MAX Graph Compiler

MAX Engine



Taking a test drive

LLAMA2 as a example

Transformer architecture:

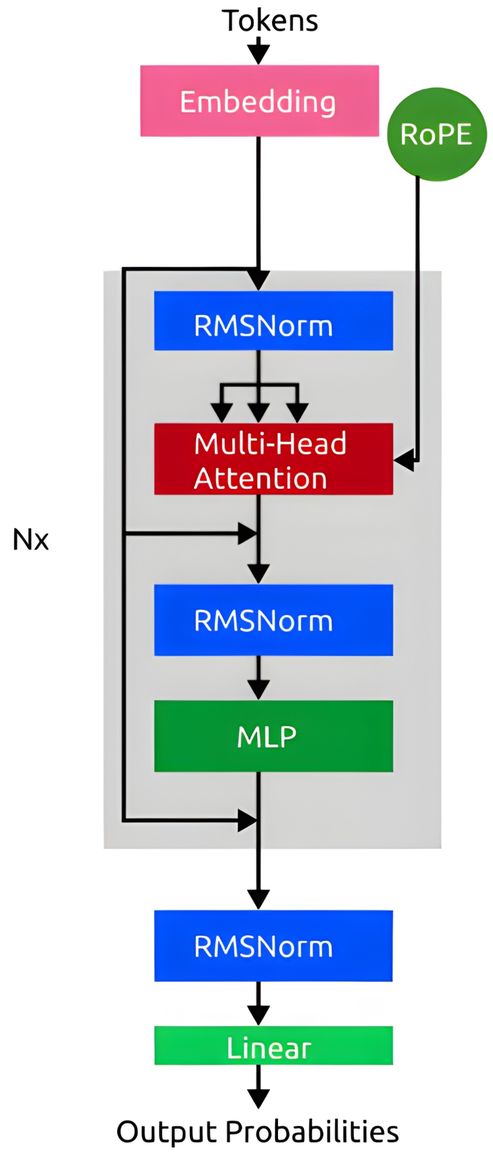
- Opened by Meta in August 2023
- Parameter sizes: 7B, 13B, and 70B
- Decoder only architecture (like GPT-3)

Dense NN algorithms:

- Multi-Head Attention
- Swish & GLU based activation
- Rotary Position Embeddings (RoPE)

Tokenization:

- Byte pair encoding



MAX Graphs provide familiar NN operators

Simple graph building APIs:

- Symbolic inputs/outputs
- Python like definition
- On the fly shape error detection

Direct control over MAX engine:

- Standard operations in the box
- Build your own abstractions

High performance + Generality

- Together at last

```
@value
struct FeedForward:
    var w1: Symbol
    var w2: Symbol
    var w3: Symbol
    def __call__(self, input: Symbol) -> Symbol:
        return (ops.silu(input @ self.w1)
                * (input @ self.w3)) @ self.w2

@value
struct RMSNorm:
    var eps: FloatLiteral
    var weight: Symbol

    def __call__(self, input: Symbol) -> Symbol:
        scale = ops.rsqrt(
                    ops.mean(input**2.0, axis=-1)
                    + self.eps)
        return input * scale * self.weight
```

Attention blocks are just more abstractions

Same principles apply up stack

Real models need customization:

- Multi-head attention
- Different embedding
- Key/value heads

The obvious code "just works"

- No complex tracing or extraction

```
def __call__(
    self,
    input: Symbol,
    start_pos: Symbol,
    freqs_cis: Symbol,
    k_cache: Symbol,
    v_cache: Symbol,
) -> SymbolTuple:
  xq = input @ self.wq
  xq = xq.reshape(batch_size, seq_len,
       self.n_heads, self.head_dim)
  xq = self.rope(xq, freqs_cis)

   ...
  scores = (xq @ keys.swap_axes(2, 3))
           * ops.rsqrt(head_dim)
  scores = scores + self.attention_mask(start_pos,
           seq_len.reshape(1))
  output = ops.softmax(scores) @ values
  output = output.swap_axes(1, 2).reshape(
           batch_size, seq_len, -1)

  return output @ self.wo, xk, xv
```

Full Transformer!

```
@value
struct TransformerBlock[dtype: DType]:
    var attention: Attention[dtype]
    var feed_forward: FeedForward
    var attention_norm: RMSNorm
    var ffn_norm: RMSNorm

    def __call__(self, input: Symbol,
        start_pos: Symbol, freqs_cis: Symbol,
        k_cache: Symbol, v_cache: Symbol,
    ) -> SymbolTuple:
        _attention_out = self.attention(
            self.attention_norm(input),
             start_pos, freqs_cis, k_cache, v_cache
        )
        attention_out = _attention_out[0]
        k_cache_update = _attention_out[1]
        v_cache_update = _attention_out[2]
        h = input + attention_out
        h = h + self.feed_forward(self.ffn_norm(h))
        return h, k_cache_update, v_cache_update
```

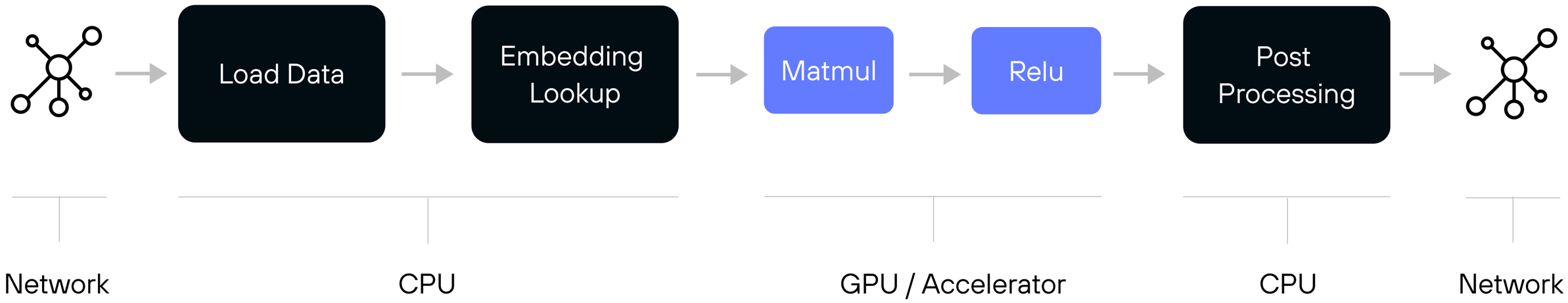
Llama2 Model

```
struct Llama2[ModelT: LoadableModel]:
  var params: ModelT
  var hyperparams: LlamaHParams

  def __init__(inout self, model_path: Path):
     self.params = ModelT(model_path)
     self.hyperparams = self.params.hyperparams()

  def build_graph(inout self, name: String):
     ...
     var layers = List[TransformerBlock]()
     for i in range(self.hyperparams.n_layers):
        var layer = TransformerBlock(
          attention=Attention(
             n_heads=self.hyperparams.n_heads,
             n_kv_heads=self.hyperparams.n_kv_heads,
             head_dim=self.hyperparams.head_dim,
             dim=self.hyperparams.dims,
             n_rep=self.hyperparams.n_rep,
             wq=self.params.get("attn_q", i),
             wk=self.params.get("attn_k", i),
             wv=self.params.get("attn_v", i),
             wo=self.params.get("attn_output", i),
             …
```

AI engine orchestrates kernels + communication



Kernels are critical, but only a small part of the overall computation:

- AI Engine picks the kernels and orchestrates how and when they are invoked
- Provides integration points with unstructured host compute

Manages distributed heterogeneous CPU+GPU compute:

- Modern systems have multiple architectures + multiple nodes

Tokenizer

Generality to the "weird" stuff:

- Unstructured code
- Linked list processing
- Other orchestration

```
while merge_options:
    merge = merge_options.pop()
    # Check whether the best merge is still valid
    if merge.left not in tokens or merge.right not in tokens:
        continue  # outdated merge option
    merged = tokens[merge.left] + tokens[merge.right]
    if len(merged) != merge.checksum:
        continue  # outdated merge option
    # Merge the right token into the left token, then
    # add any new valid merge options to the priority queue.
    left = tokens.prev(merge.left)
    right = tokens.next(merge.right)
    tokens[merge.left] = merged
    tokens.remove(merge.right)
    …
```

Compiling and running the model

```
tokenizer = BPETokenizer.from_file(TOKENIZER_PATH)
initial_prompt = "<s> I believe the fire emoji is"
prompt = tokenizer.encode(initial_prompt, bos=String("\n<s>\n"))
model = Llama2[ModelT](MODEL_PATH)
module = model.build_graph("llama_model")
session = InferenceSession()
compiled_model = session.load_model(module)
max_tokens = 256
k_cache_buff = cache_init(model, max_tokens)
v_cache_buff = cache_init(model, max_tokens)
input_map = session.new_tensor_map()
input_map.borrow("input0", tokens)
input_map.borrow("input1", k_cache_buff)
input_map.borrow("input2", v_cache_buff)
result_map = model.execute(input_map)
```

MAX

Get the full code from MAX on github:

```
git clone https://github.com/modularml/max.git
```

Optimized Kernels

Memory Planning

Novel Fusion

Device Assignment

MAX engine has many techniques

Layout Selection

Constant Folding

CUDA Streams

User Extensible Kernels

Arithmetic Simplifications

MAX

Model pipelines with MAX Serving ⚡

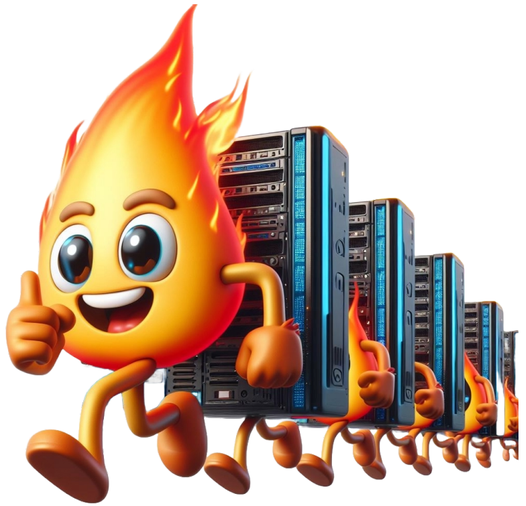
MAX ⚡ Serving

Building on industry standards:

- NVIDIA Triton Inference Server support via MAX backend
- Kubernetes orchestration
- Support for complex pipelines

Exposes the full power of the engine to serving:

- Full AI compiler power + multiple AI framework support
- Mojo APIs for op extensibility and custom logic



Customize GenAI pipelines

All the benefits of Mojo:

- Accessibility for Python devs
- Don't need C++ or Rust
- Full generality of a CPU language

Scalability to custom host logic:

- Batching and caching
- Novel tokenization approaches
- Custom preprocessing

```
alias PROMPT_LEN = 128
tokens = Tensor[DType.int64]((batch_size, PROMPT_LEN))
tokens = tokenizer.encode(prompt, bos="\n<s>\n")

cache_size = 0
for token_idx in range(PROMPT_LEN, MAX_SEQ_LEN + 1):
    attention_mask = Tensor[DType.int64](
        TensorShape(batch_size, token_idx + 1))

    result_map = execute(
        compiled_model,
        session,
        tokens,
        attention_mask,
        key_cache,
        value_cache,
    )

    key_cache = result_map.get("past_keys")
    value_cache = result_map.get("past_values")
    tokens = argmax(result_map.get("logits"))

    print_no_newline(tokens[Index(0, -1)], ",")
```

MAX Deployment

```
curl https://get.modular.com | sh - &&
modular install max
```

```
max build --container your_llm_serve.mojo
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: serving
  labels:
    app: serving
spec:
  replicas: 3
  containers:
  - name: your_llm_serve
    image: xn--4v8h@sha256:46903a36
```

your_llm_serve.mojo

```
struct Service:
    var model: Model
    var tokenizer: ITokenizer
    var kv_cache: IKVCacheManager
    var execute_batch: ContinuousBatch
    def __init__(inout self,
                 model_path: String,
                 tokenizer: ITokenizer,
                 kv_cache: IKVCacheManager):
        session = engine.InferenceSession()
        …
        fn execute(input: TensorMap) -> TensorMap:
            return self.model.execute(input)
        self.execute_batch = ContinuousBatch[
              execute, …](self.kv_cache)
    async def handle(self, request: Request,
                     response: Response):
        prompt = await self.tokenize_batch(…)
        for token in self.execute_batch.stream(prompt):
            detokenized_token =
                 await self.detokenize_batch(token)
            response.StreamInstructResponse(
                     detokenized_token)
        response.Done()
```

Wrapping up...

Programmable
Performant
Portable

MAX

Unify AI Development and Deployment

Traditional AI

Gen AI

Roadmap ahead

2024Q1: Next steps in Mojo OSS!

2024Q2:

- MAX for macOS 👨‍💻
- Commercially supported on AWS
- Many Mojo improvements + libraries
- Initial NVIDIA GPU Support

Beyond:

- Many, many, things going on...

MAX

Download now, completely **free**!

modul.ar/max

Let's unlock the next generation together!

Questions?