Modular

Mojo + PyTorch:
A Simpler Path to Custom Kernels

Spenser Bauman
Compiler Engineer @
MODULAR INC.

Oct 22, 2025

Agenda

01 **Why Mojo & MAX?**

02 **Write a custom kernel in Mojo**

03 **Interface to it easily from PyTorch**

04 **Good performance without the pain**

❌**What you won't see:**

Hardware lock-in

PyBind, CMake, setuptools

Performance gaps

CUDA version mismatches

Library path problems

Why Mojo?

A look at existing performance libraries

Many are written in assembly…

… so much for abstraction even across members of an architectural family …

```
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

```
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

```
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

Template (*but not templates*) 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

Pervasive suffering

Maintainability,
debugging, tooling, …

Choose: Performance *or* Ease of Use

Why MAX?

It started with Frameworks with Large Kernel Libraries

Orthogonal Kernels

Hand Fused Kernels

Thousands of kernels later: it doesn't scale!

Then: AI Compilers To Rescue?

There's a problem … generality!

But: Where is Generality?

Many common limitations…

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

GenAI changed the game:

- Continuous innovation in attention and other mechanisms: more explosion of fused kernels
- Diversity of KVCache optimizations
- Performance/TCO is critical for inference

"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

Challenge: Compiler Scalability!

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"!

To the rescue

MAX - Modular Accelerated eXecution

- 🔥 **Mojo Programming Language**: System-level performance with Python syntax
- 📊 **Python Graph API**: Familiar PyTorch-like interface
- ⚙️ **Graph Compiler**: Automatic optimization and fusion
- 💎 **Optimized GPU Kernels**: Hardware-specific optimizations
- 🔧 **Hardware Abstraction**: Write once, run anywhere

Why Mojo🔥 & MAX® ?

A Pythonic System Programming Language

- Modern high-level systems language
- Hardware agnostic (CPU and GPU)
- Memory safety, powerful metaprogramming system
- Full toolchain + VS Code + LSP support
- Best of both worlds: productivity AND performance

Cross-Platform Hardware Support

- Supports CPU and GPU targets without vendor lock-in
- Works with Nvidia, AMD (Apple Silicon in nightlies)
- Significantly smaller container sizes vs competitors (600MB vs 11GB)

Write a custom kernel in Mojo

Anatomy of a Mojo Kernel

```
@compiler.register("grayscale")
struct Grayscale:
    @staticmethod
    fn execute[
        target: StaticString,
    ](
        img_out: FusedOutputTensor[dtype=float32, rank=2],
        img_in: InputTensor[dtype=float32, rank=3],
        ctx: DeviceContextPtr,
    ) raises:
        @parameter
        fn color_to_grayscale[
            width: Int
        ](idx: IndexList[img_out.rank]) -> SIMD[float32, width]:
            var row, col = idx[0], idx[1]
            var r = img_in.load[width](Index(0, row, col))
            var g = img_in.load[width](Index(1, row, col))
            var b = img_in.load[width](Index(2, row, col))
            return 0.21 * r + 0.71 * g + 0.07 * b

        foreach[color_to_grayscale, target=target](img_out, ctx)
```

Registered name

Entry point

Structured Inputs (not pointers)

Fusion Properties

Parallelize/Vectorize across CPU or GPU

And how does it perform?

Image size: 16000x16000

What about something more realistic?

At Modular, all our kernels are implemented in Mojo

🧑‍🚀... by a relatively small team of engineers (~10 people),

🛠️... for all the hardwares we support: CPUs + 7 major GPU architectures from 3 different vendors (NV, AMD, Apple),

🚀... and get SOTA throughput on production models,

🔥... so how do we do on performance critical kernels?

Mojo vs cuBLAS (B200)

Matrix Multiplication

2 APIs to Integrate Mojo into PyTorch

1. 👟 An API to directly call custom kernels
2. 📈 An API to call full MAX graphs

Modular

Call Your Kernel from PyTorch

Load your library of custom kernels (no compile step):

```
custom_ops = CustomOpLibrary("path/to/ops")
```

Wrap the destination passing kernel:

```
@torch.compile
def max_grayscale(x: torch.Tensor) -> torch.Tensor:
    result = torch.empty(x.shape[1:], dtype=x.dtype, device=x.device)
    custom_ops.grayscale(result, x)
    return result
```

Compatible with “fullgraph=True”

When to use this API

Tradeoffs for this API:

✅Simple for a small number of kernels

❌Overhead due to __dlpack__ & stream synchronization

❌Introduced optimization boundaries (no fusion)

Integrate with MAX Graphs

This API:

❌More work for single ops

✅Scales beyond a single operation

✅Less marshalling/dispatch overhead

✅More optimization opportunities

```python
@max.torch.graph_op(custom_ops)
def max_grayscale_graph(
    pic: max.graph.TensorValue
) -> max.graph.TensorValue:
    c, h, w = pic.shape

    scale = ops.constant(
        np.array([0.21, 0.71, 0.07]),
        dtype=DType.float32,
        device=DeviceRef.GPU()
    ).reshape([c, 1, 1])

    scaled = pic * ops.broadcast_to(scale, (c, h, w))
    grayscaled = ops.sum(scaled, axis=0)
    # max reductions don't remove the dimension, need to squeeze
    return ops.squeeze(grayscaled, axis=0)
```

Key Takeaways

🐍 + 🔥 = 😎

1. Prototype and optimize kernels in one system
2. No vendor lock-in
3. 🆕 Easy integration with PyTorch

Modular

Thank You

Custom ops tutorial
modul.ar/ptc-ops

Support & discussion
forum.modular.com

Optimizing Matmul on Blackwell:
https://www.modular.com/matrix-multiplication-on-blackwell