15-442/15-642: Machine Learning Systems

Universal Large-language Model Deployment with ML Compilation

Spring 2024

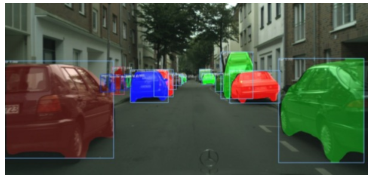
Tianqi Chen and Zhihao Jia

Carnegie Mellon University

History of Machine Learning Revolutions

Big Data

Deep Learning



Generative AI

Bard Llama 2

Key Capabilities

Recommendation
Data analytics

Strong pattern recognition capabilities

Open ended conversations
Generalist models

ML Systems

dmlc XGBoost

Apache Spark

GraphLab

TensorFlow

tvm

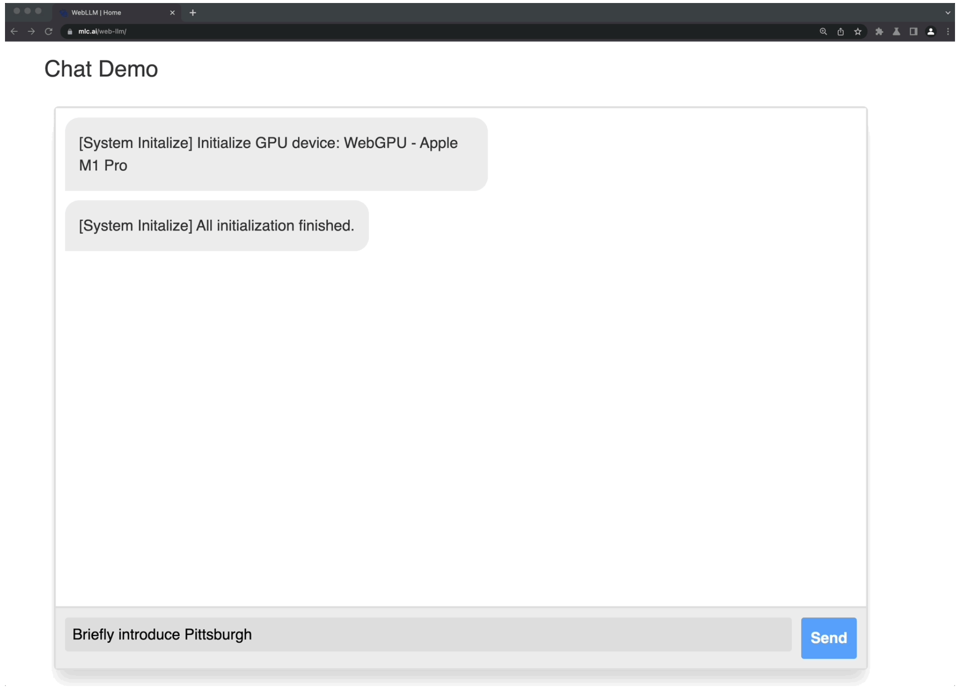
MLSys plays an even more central role

2010 **2013** **2023**

Systems for Generative AI: Challenges and Opportunities

Generative AI

Open ended conversations
Generalist models



Memory Llama-70B would consume 320GB VRAM to just to store parameters in fp32

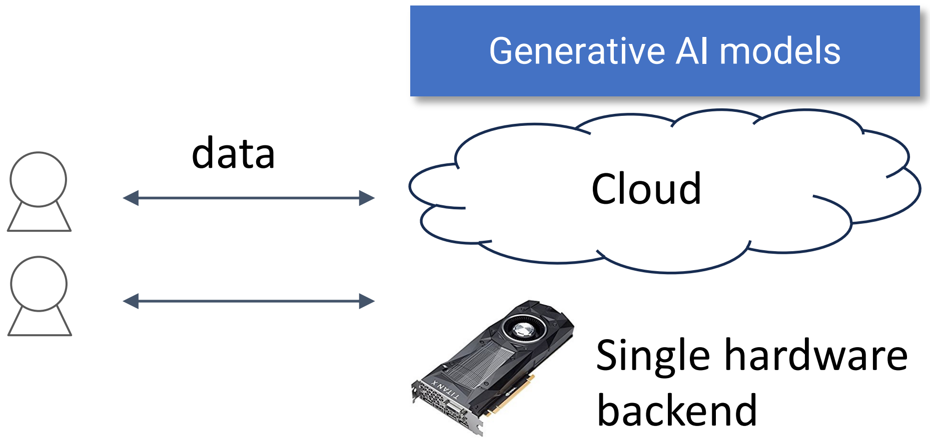
Compute The post-Moore era brings great demand for diverse specialized compute, system support becomes bottleneck

Integration Goes beyond single chat model, modern AI applications can see, talk, compose music. Need to coordinate multiple models and system components.

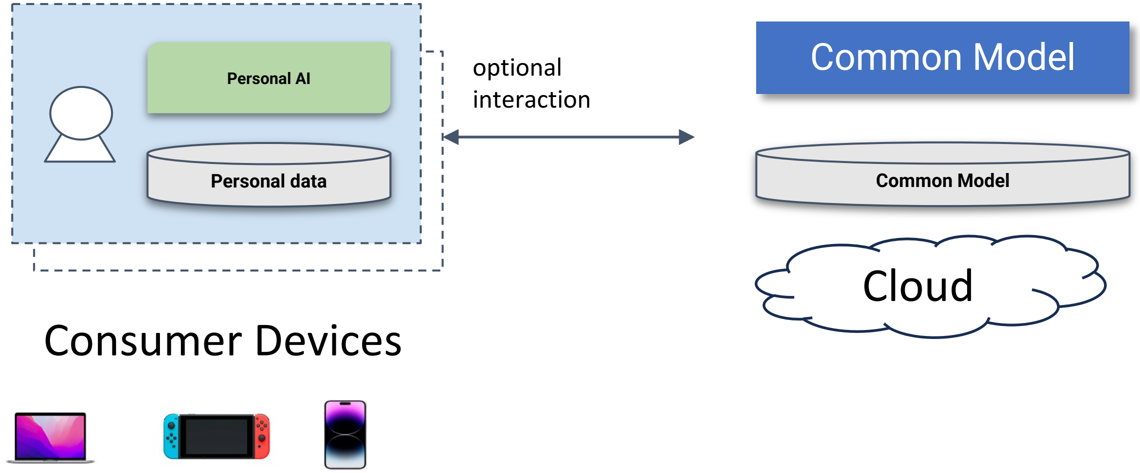
Evolutions and co-design Keep up with new demands, new modeling approaches, hardware variants, and co-design

The Case for Bringing Generative AI Everywhere

Generative AI Paradigm Today



Just like personal computers
can we get our own personal AI?



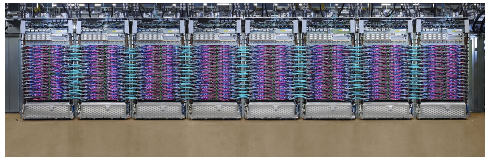
Machine Learning Systems: Typical Engineering Approach

ML Models

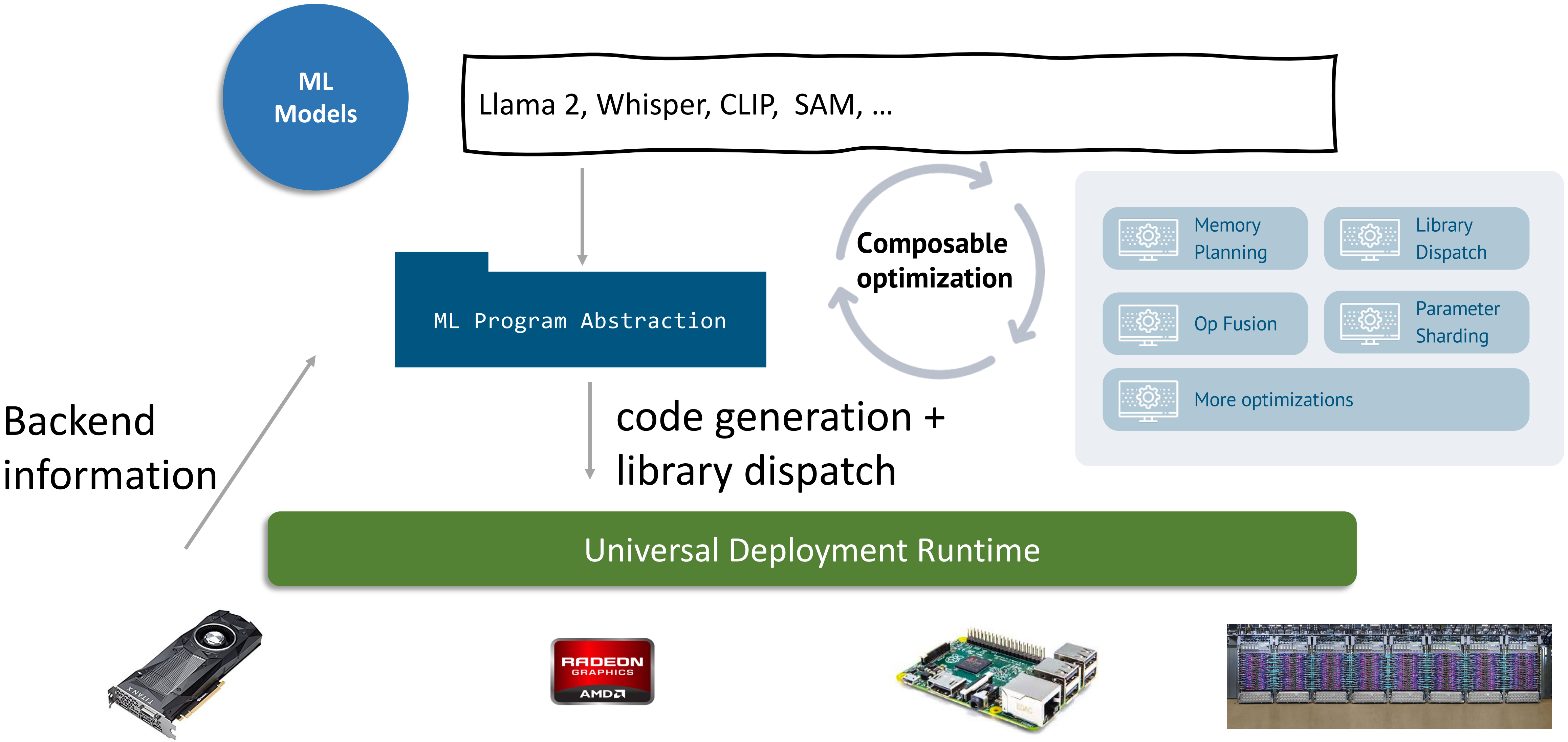
Llama 2, Whisper, CLIP, SAM, …

- Specialized libraries and systems for each backend (labor intensive)
- Non-automatic optimizations

Nvidia Stack	AMD Stack	ARM-Compute	TPU Stack

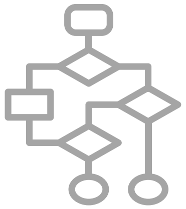
ML Compilation



Abstractions for ML Compilation

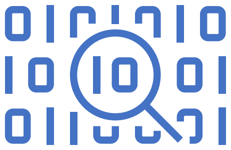
There are four different categories of abstractions we use to accelerate machine learning today

Computational Graphs



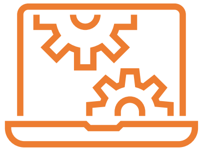
Computational graph and its extensions enable high level program rewriting and optimization.

Tensor Programs



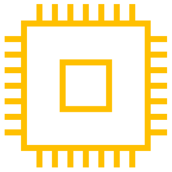
Tensor program abstractions focus on loop and layout transformation for fused operators.

Libraries and Runtimes



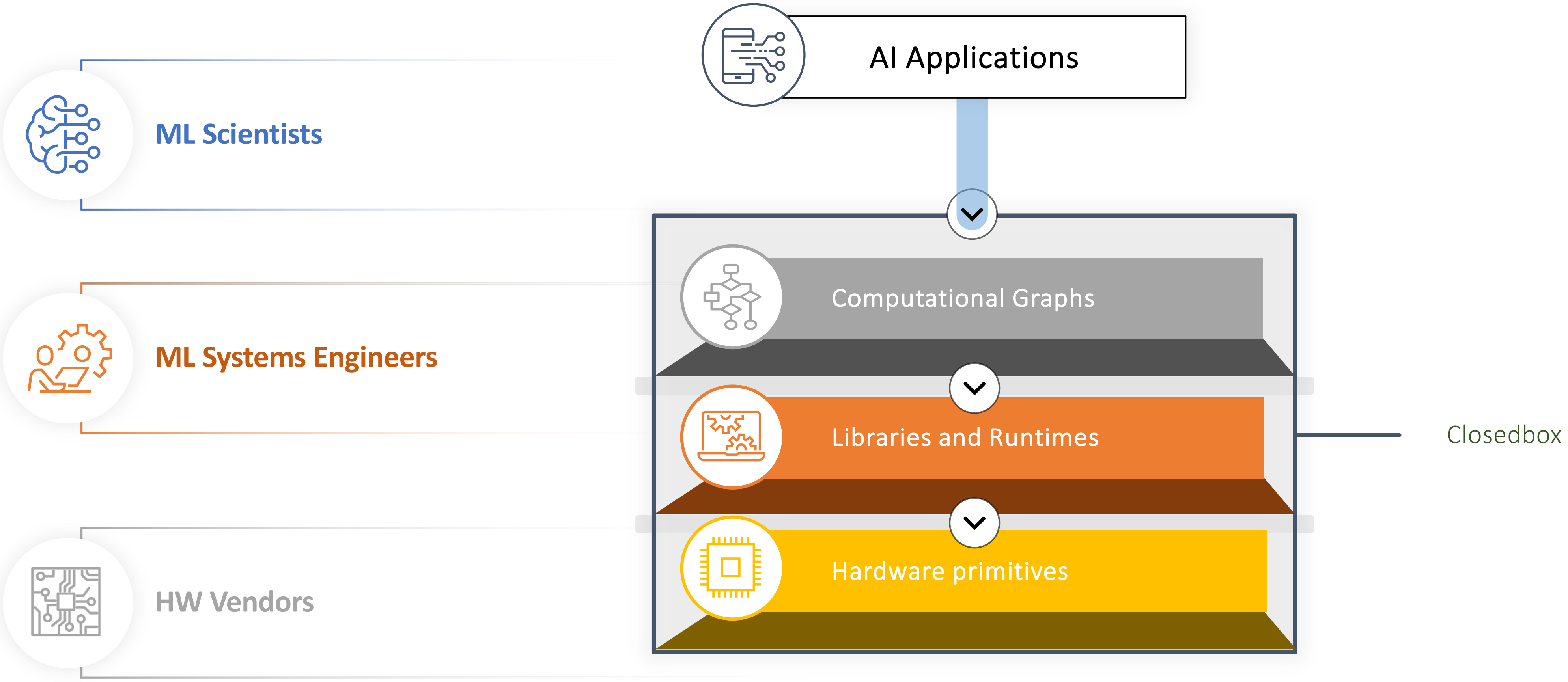
Optimizing libraries are built by vendors and engineers to accelerate key operators of interest.

Hardware Primitives

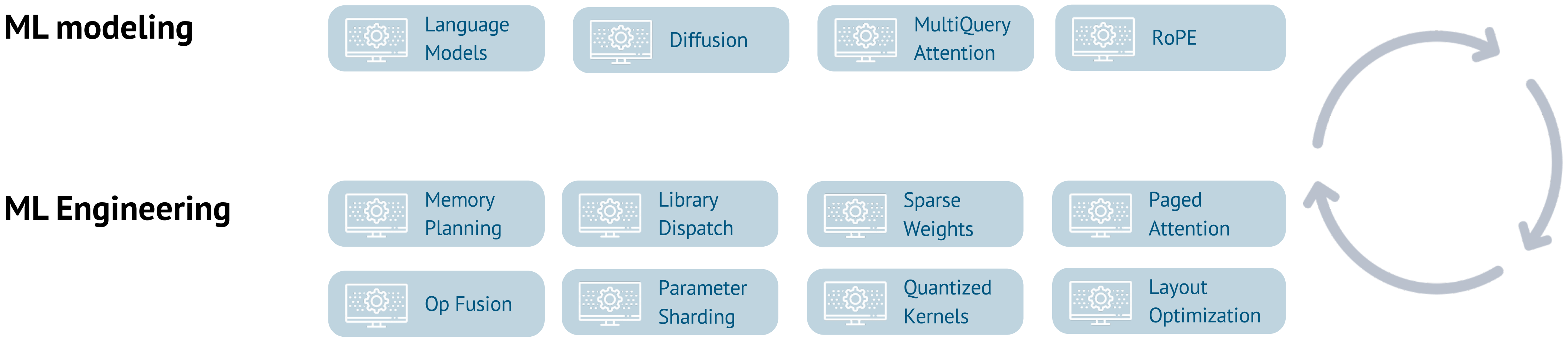


The hardware builders exposes novel primitives to provide native hardware acceleration.

Current Frameworks and Challenges



What is the Biggest Challenge?



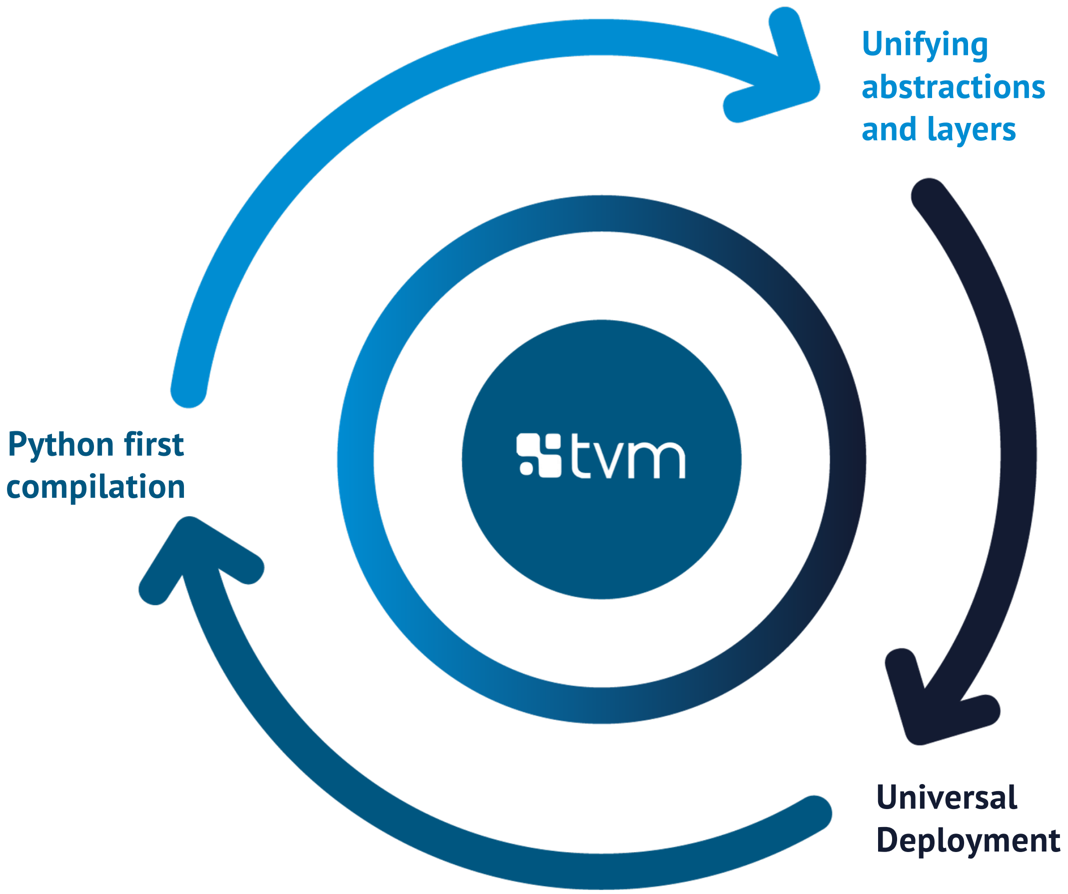
ML engineering now becomes critical and go hand in hand with ML modeling
It is not about build silver bullet once but **continuous improvement and innovations**

TVM Unity

Mission

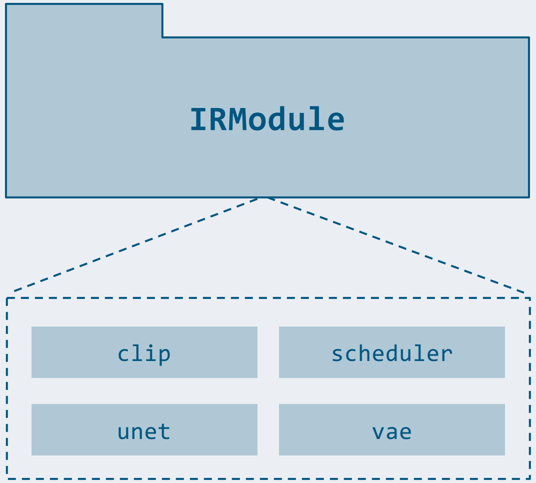
Empower community members to optimize any machine learning models and run them on any hardware backend.

This is not a single step journey.



IRModule as the Central Abstraction

Centers around one key construct



A collection of (tensor) functions that correspond to model components.

```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class Module:
    @R.function
    def vae(
        data: R.Tensor(("n", 4, 64, 64), "float32"),
        params: R.Tuple(R.Tensor((4, 4, 1, 1), "float32"),
                        R.Tensor((1, 4, 1, 1), "float32"),
                        ...)
    ) -> R.Tensor(("n", 512, 512, 3),"float32"):
        n = T.int64()
        with R.dataflow():
            w0: R.Tensor((4, 4, 1, 1), "float32") = params[0]
            lv0: R.Tensor((n, 4, 64, 64), "float32") = R.nn.conv2d(
                data, w0, strides=[1, 1]
            )
            b0: R.Tensor((1, 4, 1, 1), "float32") = params[1]
            lv1: R.Tensor((n, 4, 64, 64), "float32") = R.add(lv0, b0)
            ...
```

Accessible in python through TVMScript

```
>>> mod.show()
```

Unifying abstractions by encapsulation computational graph, tensor program, library, hardware primitives, and their interactions in the same module

Python First Development

Import

```python
mod = frontend.from_fx(torch_graph)
```

Inspect and interact

```python
mod = my_script_module.Module

sch = tvm.tir.Schedule(mod)
sch.work_on("add")
add_block = sch.get_block("T_add")
(i,) = sch.get_loops(add_block)
i0, i1 = sch.split(i, [None, 128])
sch.bind(i0, "blockIdx.x")
sch.bind(i1, "threadIdx.x")
mod = sch.mod

mod.show()
```

tvm

IRModule

Python first API for productive and accessible developments through all stages of the stack.

Transform and optimize

```python
seq = transform.Sequential([
    transform.FuseOps(),
    transform.FuseTIR()
])
mod = seq(mod)
```

Deploy

```python
ex = relax.build(mod, target)
ex.export_library("model.so")
```

Universal Deployment

IRModule

```
@tvm.script.ir_module
class Module:
    @R.function
    def vae(
        data: R.Tensor(("n", 4, 64, 64), "float32"),
        params: R.Tuple(R.Tensor((4, 4, 1, 1), "float32"),
                        R.Tensor((1, 4, 1, 1), "float32"),
                        ...)
    ) -> R.Tensor(("n", 512, 512, 3),"float32"):
        n = T.int64()
        with R.dataflow():
            w0: R.Tensor((4, 4, 1, 1), "float32") = params[0]
            lv0: R.Tensor((n, 4, 64, 64), "float32") = R.nn.conv2d(
                   data, w0, strides=[1, 1]
             )
            b0: R.Tensor((1, 4, 1, 1), "float32") = params[1]
            lv1: R.Tensor((n, 4, 64, 64), "float32") = R.add(lv0, b0)
            ...
```

```
>>> ex = relax.build(mod, target)
```

Every tensor function (e.g. vae) becomes a native runnable function on the target platform after build.

Runs everywhere

Python

```
data = tvm.nd.from_dlpack(other_array)
vm = relax.VirtualMachine(ex, tvm.cuda())
out = vm["vae"](data, params)
```

torch.compile integration

```
vae = torch.compile(
    vae, backend=relax.frontend.relax_dynamo())
out = vae(data, params)
```

c++

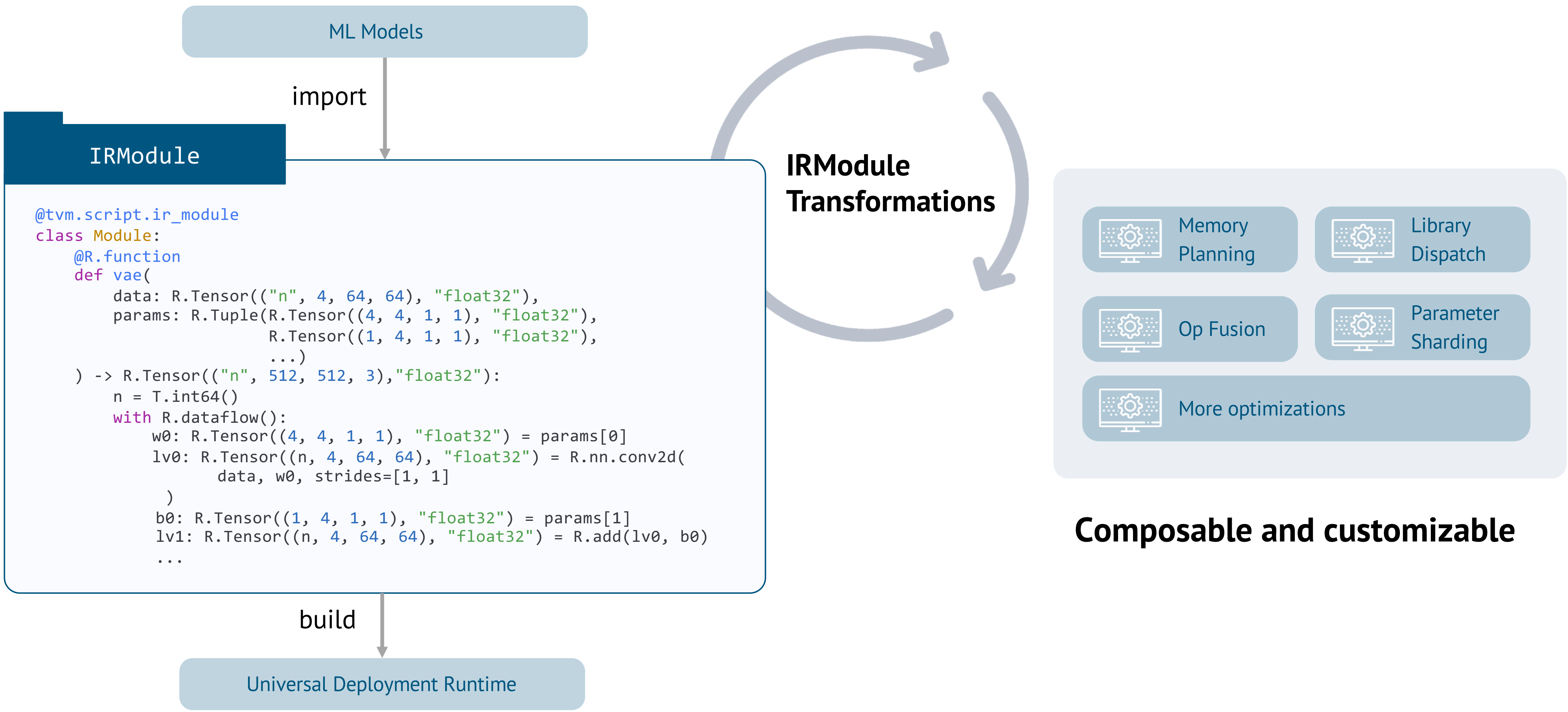
```
runtime::Module vm = ex.GetFunction("load_executable")()
vm.GetFunction("init")(...)
NDArray out = vm.GetFunction("vae")(data, params)
```

Javascript (web)

```
tvm = await tvmjs.instantiate(wasmSource, new EmccWASI())
vm = tvm.createVirtualMachine(tvm.webgpu())
out = vm.getFunction("vae")(data, params)
```

More platforms with tvm runtime.

Productive Framework for ML Compilation



Composable and customizable

Continuous Improvement Process

ML Compilation Out of Box

Model

↓

ML Compilation

↓

Optimized Model

Continuous ML Compilation Evolution Process

- New hardware backends
- New settings
(e.g. dyn shape, distributed)
- New libraries
- New joint optimizations

This is not a one shot game, but continuous ML compilation evolution process for every new model, backend features, new improvements. We can enable more people to do it, together :)

Elements of TVM Unity

Abstraction Elements of TVM Unity

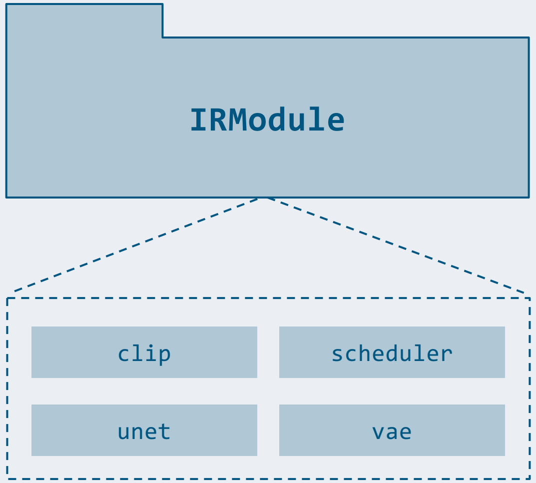
First-class symbolic shape support

Composable Tensor Program Optimization

Unifying Libraries and Compilation

First class Symbolic Shape

Centers around one key construct



A collection of (tensor) functions that correspond to model components.

Accessible in python through TVMScript

```
>>> mod.show()
```

```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class Module:
    @R.function
    def vae(
        data: R.Tensor(("n", 4, 64, 64), "float32"),
        params: R.Tuple(R.Tensor((4, 4, 1, 1), "float32"),
                        R.Tensor((1, 4, 1, 1), "float32"),
                        ...)
    ) -> R.Tensor(("n", 512, 512, 3),"float32"):
        n = T.int64()
        with R.dataflow():
            w0: R.Tensor((4, 4, 1, 1), "float32") = params[0]
            lv0: R.Tensor((n, 4, 64, 64), "float32") = R.nn.conv2d(
                data, w0, strides=[1, 1]
            )
            b0: R.Tensor((1, 4, 1, 1), "float32") = params[1]
            lv1: R.Tensor((n, 4, 64, 64), "float32") = R.add(lv0, b0)
            ...
```

First-class symbolic shape support to enable dynamic shape compilation.

Symbolic Shape vs Any Shape

Symbolic Shape

```
@R.function
def symbolic_shape_fn(x: R.Tensor(("n", 2, 2), "float32")):
    n, m = T.int64(), T.int64()
    with R.dataflow():
        lv0: R.Tensor((n, 4), "float32")) = R.reshape(x, R.shape(n, 4))
        lv1: R.Tensor((n * 4,), "float32")) = R.flatten(lv0)
        lv2: R.Tensor(ndim=1, dtype="float32") = R.unique(lv1)
        lv3 = R.match_cast(lv2, R.Tensor((m,), "float32"))
        gv0: R.Tensor((m,), "float32") = R.exp(lv3)
        R.output(gv0)
    return gv0
```

- Tracks the shape values (n, n * 4)
- More optimizations
- Flexible fallback for unknown and rematch
- Shape is part of computation

Any Shape Dimension

```
@R.function
def any_shape_fn(x: R.Tensor((?, 2, 2), "float32")):
    n = R.get_shape_value(x, axis=0)
    with R.dataflow():
        lv0: R.Tensor((?, 4), "float32")) = R.reshape(x, R.shape(n, 4))
        lv1: R.Tensor((?,), "float32")) = R.flatten(lv0)
        lv2: R.Tensor(?, "float32") = R.unique(lv1)

        gv0: R.Tensor((?,), "float32") = R.exp(lv3)
        R.output(gv0)
    return gv0
```

- Most approaches so far
- ? denotes any shape value
- No relation information: cannot prove shape equivalence by only looking at any dimensions

Optimizations Enabled by Symbolic Shape

Static memory planning for dynamic shape

Dynamic shape aware operator fusion

Layout rewriting and padding

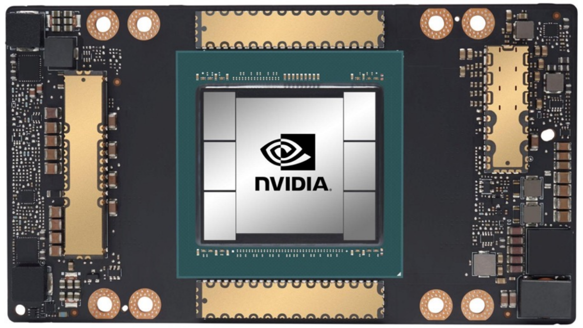
Abstraction Elements of TVM Unity

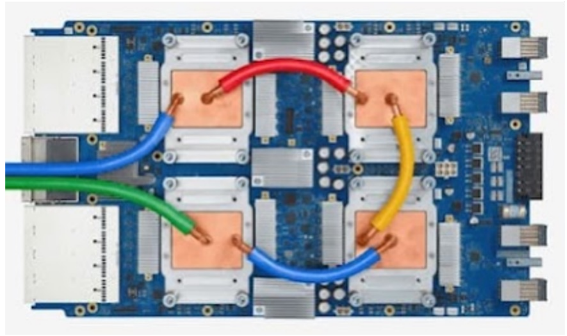
First-class symbolic shape support

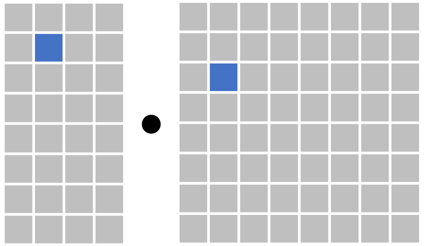
Composable Tensor Program Optimization

Unifying Libraries and Compilation

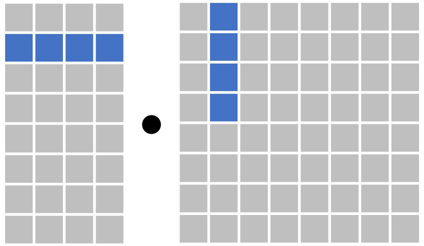
Hardware Trend



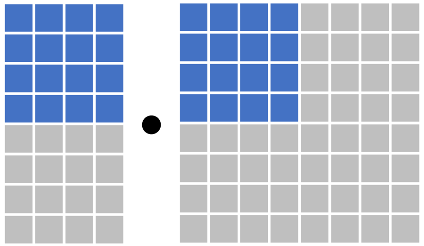




Scalar Computing



Vector Computing



Tensor Computing

- Google TPU
- Nvidia Tensor Core
- AMD Matrix Core
- Intel Matrix Engine
- Apple Neural Engine
- Arm Ethos-N
- T-Head Hanguang
- ……

Elements of a Tensorized Program

```
for ic.outer, kh, ic.inner, kw in grid(...):

  for ax0 in range(...):
    load_matrix_sync(A.wmma.matrix_a, 16, 16, 16, ...)


  for ax0 in range(...):
    load_matrix_sync(W.wmma.matrix_b, 16, 16, 16, ...)


  for n.c, o.c in grid(...):
    wmma_sync(Conv.wmma.accumulator,
              A.wmma.matrix_a,
              W.wmma.matrix_b,
              ...)


for n.inner, o.inner in grid(...):
  store_matrix_sync(Conv.wmma.accumulator, 16, 16, 16)
```

Example Snippet: Conv2D on Tensor Core

Optimized loop nests with thread binding

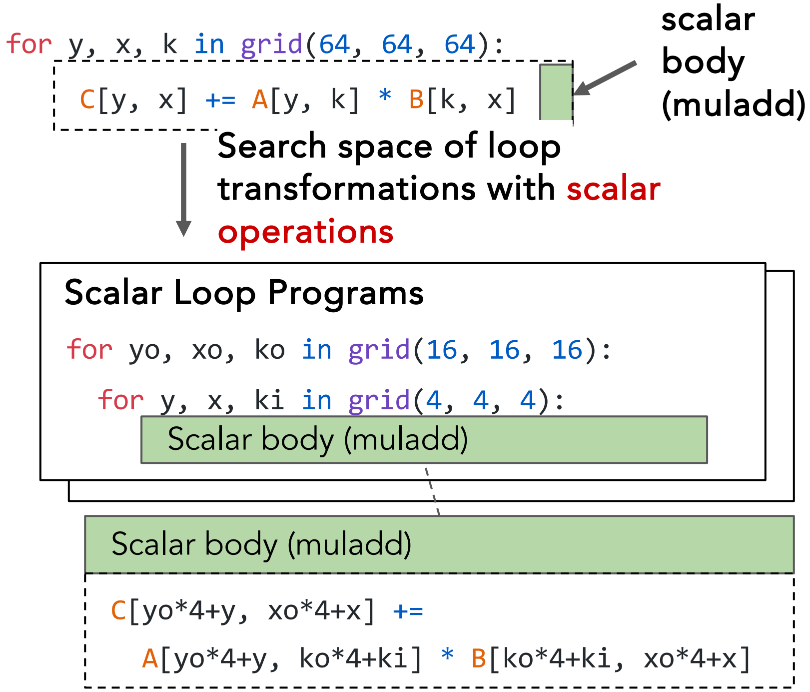
Multi-dimensional data load into specialized hardware storage

Opaque tensorized computation body 16x16 matrix multiplication

Multi-dimensional data store

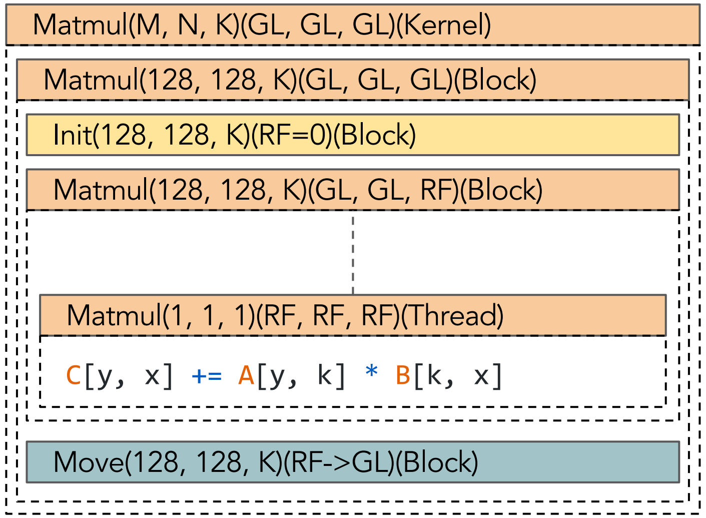
Existing Abstractions

Bottom up: Transform and optimize multi-dimensional loop nests with scalar body (Halide, TVM/TE, Affine)



Harder to represent tensorized computation body

Top Down: Recursive decomposition of tasks into smaller ones (Fireiron, Stripe)



Less obvious for loop nest transformation optimizations

TensorIR Abstraction: Divide and Solve(Conquer)

```
for y, x, k in grid(64, 64, 64):
  C[y, x] += A[y, k] * B[k, x]
```

Introduce a key abstraction called block to divide and isolate the problem space into outer loop nests and tensorized body

```
for yo, xo, ko in grid(16, 16, 16):
  block (by=yo, bx=xo, bk=ko)
    for y, x, k in grid(4, 4, 4):
      C[by*16+y, bx*16+x] +=
        A[by*16+y, bk*16+k] * B[bk*16+k, bx*16+x]
```

Tensorized body (matmul4x4) isolated from the outer loop nests

Key Ideas

- Divide problem into sub-tensor computation blocks
- Generalize loop optimization for tensorized computation
- Combination of the above approaches in any order

Search space of loops transformations with tensorized operations

Tensorized Programs

```
for yo, xo, k in grid(4, 4, 16):
  for yi, xi in grid(4, 4):
    block (by, bx, bk=...)
    Tensorized body (matmul4x4)
```

Map tensorized body based on instructions provided by the backend.

Option 0: Tensorized body (matmul4x4)

```
accel.matmul_add4x4(
  C[by*16:by*16+4, bx*16:bx*16+4],
  A[by*16:by*16+4, bk*16:bk*16+4],
  B[bk*16:bk*16+4, bx*16:bx*16+4])
```

Option 1: Tensorized body (matmul4x4)

```
for y, x, k in grid(4, 4, 4):
  C[by*16+y, bx*16+x] +=
    A[by*16+y, bk*16+k] * B[bk*16+k, bx*16+x]
```

Example

Computation: `C = exp(A + 1)`

TVMScript

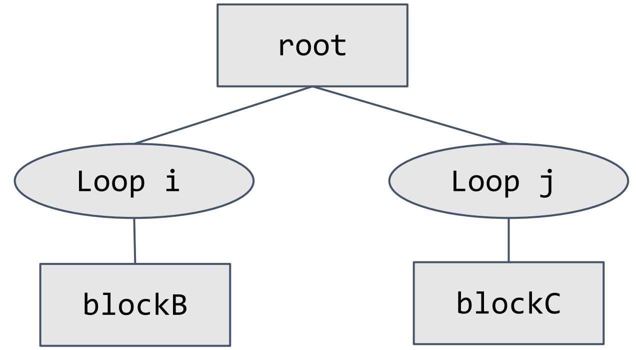
```
@tvm.script.tir
def fuse_add_exp(a: ty.handle, c: ty.handle) -> None:
    A = tir.match_buffer(a, (64,))
    C = tir.match_buffer(c, (64,))
    B = tir.alloc_buffer((64,))

    for i in tir.grid(64):
        with tir.block([64], "blockB") as [vi]:
            vi = tir.bind(i)
            B[vi] = A[vi] + 1

    for j in tir.grid(64):
        with tir.block([64], "blockC") as [vi]:
            vi = tir.bind(j)
            C[vi] = exp(B[vi])
```

Multi-dimensional **buffer**

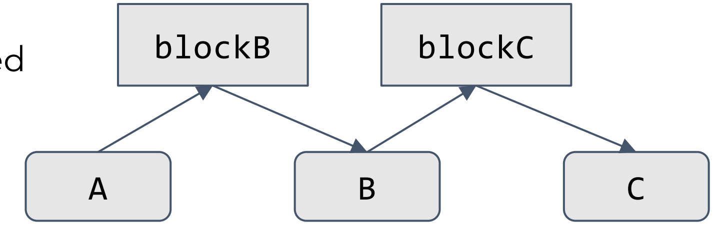
Loop nests

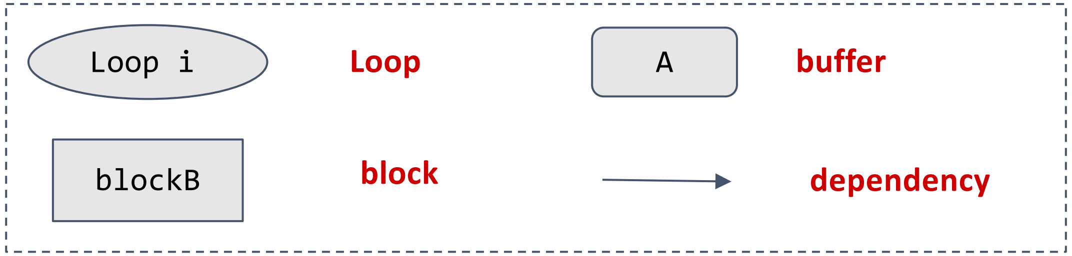
Computational **block**

AST



Producer consumer dependencies (inferred from AST)





Elements of TensorIR: Tensorized Computation

```
@tvm.script.tir
def fuse_add_exp(a: ty.handle, c: ty.handle) -> None:
   A = tir.match_buffer(a, (64,))
   C = tir.match_buffer(c, (64,))
   B = tir.alloc_buffer((64,))

   for i in tir.grid(8):
       with tir.block([8], "blockB") as [vi]:
           vi = tir.bind(i)
           tir.reads(A[vi * 8: vi * 8 + 8])
           tir.writes(B[vi * 8: vi * 8 + 8])
           for k in tir.grid(8):
               B[vi * 8 + k] = A[vi * 8 + k] + 1

   for j in tir.grid(64):
       with tir.block([64], "blockC") as [vi]:
           vi = tir.bind(j)
           C[vi] = exp(B[vi])
```

Block representing vectorized/tensorized computation
Add 8 elements at a time

Tensorized computation as the first class citizen

Elements of TensorIR: Block

```
for yo, xo, ko in grid(16, 16, 16):
  with block(domain=(16, 16, reduce_axis(16)),
             other_signatures) as vy, vx, vk:

    vy = var_bind(yo)
    vx = var_bind(xo)
    vk = var_bind(ko)


    for yi, xi, ki in grid(4, 4, 4):
      C[vy*4 + yi, vx*4 + xi] +=
        A[vy*4 + yi, vk*4 + ki] * B[vk*4 + ki, vx*4 + xi]
```

outer loop

signature

iterator binding

body

Block Signature

Iterator domain and constraints:

```
vy: data_parallel_axis(length=16)
vx: data_parallel_axis(length=16)
vk: reduce_axis(length=16)
```

Producer consumer dependency relations

```
read A[vy*4:vy*4+4, vk*4:vk*4+4]
read B[vk*4:vk*4+4, vx*4:vx*4+4]
reduce_update C[vy*4:vy*4+4, vx*4:vx*4+4]
```

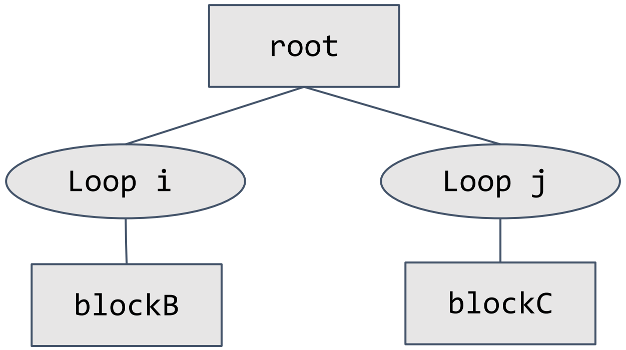
Isolate the internal computation tensorized computation from external loops

Block as Schedulable Unit

```
@tvm.script.tir
def fuse_add_exp(a: ty.handle, c: ty.handle) -> None:
    A = tir.match_buffer(a, (64,))
    C = tir.match_buffer(c, (64,))
    B = tir.alloc_buffer((64,))

    for i in tir.grid(8):
        with tir.block([8], "blockB") as [vi]:
            vi = tir.bind(i)
            tir.reads(A[vi * 8: vi * 8 + 8])
            tir.writes(B[vi * 8: vi * 8 + 8])
            my_fancy_vector_addone(A, B, 8)

    for j in tir.grid(64):
        with tir.block([64], "blockC") as [vi]:
            vi = tir.bind(j)
            C[vi] = exp(B[vi])
```

Root block: schedulable

blockB: not scheduable



A block is **schedulable** if it only contains loop nests and sub-blocks as its leaf.

We can transform the loop nests in a schedulable block

Imperative Schedule Transformation

```
for i, j in  grid(64, 64):
    produceA
    A [i, j] = …
for yo, xo, k in grid(4, 4, 16):
    for yi, xi in grid(4, 4):
        blockB
        vy = var_bind(yo*4 + yi)
        vx = var_bind(xo*4 + xi)
        vk = var_bind(ko)
        body
```

```
s = tvm.tir.Schedule(myfunc)
prodA = s.get_block("produceA")
k = s.get_loop("k")

s.compute_at(prodA, k)
```

blockB signature

Iterator domain and constraints:

```
vy: data_parallel_axis(length=16)
vk: data_parallel_axis(length=16)
vk: reduce_axis(length=16)
```

Producer consumer dependency relations

```
read A[vy*4:vy*4+4, vk*4:vk*4+4]
read B[vk*4:vk*4+4, vx*4:vx*4+4]
reduce_update C[vy*4:vy*4+4, vx*4:vx*4+4]
```

Block signature dependency information used during transformation

Imperative Schedule Transformation

```
for yo, xo, k in grid(4, 4, 16):
    for i, j in  grid(16, 4):
        produceA
        A [yo*16 + i, k*4 + j] = ...
    for yi, xi in grid(4, 4):
        blockB
        vy = var_bind(yo*4 + yi)
        vx = var_bind(xo*4 + xi)
        vk = var_bind(ko)
        body
```

```
s = tvm.tir.Schedule(myfunc)
prodA = s.get_block("produceA")
k = s.get_loop("k")

s.compute_at(prodA, k)
```

- **Interactive**: Schedule as imperative transformations of the IR.
- **Modularize**: Analysis only depend on the block signature
- **Extensible**: No schedule tree, easy to add new schedule primitives

Isolating Tensorized Computations

```
for i, j, ko in grid(64, 64, 16):
    for ki in range(4):
        block (vi = i, vj = j, reduce vk = ko*4 + ki)
        C[vi, vj] += A[vi, vk] * B[vk, vj]
```

```
s = tvm.tir.Schedule(myfunc)
ki = s.get_loop("ki")

s.blockize(ki)
```

Isolating Tensorized Computations

```
for i, j, ko in grid(64, 64, 16):
  block
  for ki in range(4):
    block (vi = i, vj = j, reduce vk = ko*4 + ki)
      C[vi, vj] += A[vi, vk] * B[vk, vj]
```

```
s = tvm.tir.Schedule(myfunc)
ki = s.get_loop("ki")

s.blockize(ki)
```

Tensorization

```
for y, x, k in grid(64, 64, 64):
    C[y, x] += A[y, k] * B[k, x]
```

Step 1. Original workload

```
for yo, xo, ko in grid(16, 16, 16):
  block (by=yo, bx=xo, bk=ko)
  for y, x, k in grid(4, 4, 4):
    C[by*16+y, bx*16+x] +=
      A[by*16+y, bk*16+k] * B[bk*16+k, bx*16+x]
```

Step 2.
Split + Reorder + Blockize
Getting the 4x4x4 matrix multiplication to be tensorized

Step 3. Substitute the inner block with equivalent computation block

Tensorized Programs

```
for yo, xo, ko in grid(16, 16, 16):
  block (by=yo, bx=xo, bk=ko)
  Tensorized body (matmul4x4)
```

Map tensorized body based on instructions provided by the backend.

Option 1: Utilize accelerator tensor instruction	Option 2: Scalar Computing

`accel.matmul_add4x4(`
`  C[by*16:by*16+4, bx*16:bx*16+4],`
`  A[by*16:by*16+4, bk*16:bk*16+4],`
`  B[bk*16:bk*16+4, bx*16:bx*16+4])`
	`for y, x, k in grid(4, 4, 4):`
	`  C[by*16+y, bx*16+x] +=`
	`    A[by*16+y, bk*16+k] *`
	`    B[bk*16+k, bx*16+x]`

Bringing TensorIR into TVM Unity

IRModule

```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class IRModule:
    @T.prim_func
    def mm(
        X: T.Buffer(("n", 128), "float32"),
        W: T.Buffer((128, 64), "float32"),
        Y: T.Buffer(("n", 64), "float32")
    ):
        n = T.int64()
        for i, j, k in T.grid(n, 64, 128):
            Y[i, j] += X[i, k] * W[k, j]

    @R.function
    def main(
        X: R.Tensor(("n", 128), "float32"),
        W: R.Tensor((128, 64), "float32")
    ):
        n = T.int64()
        with R.dataflow():

            lv0 = R.call_tir(mm, (X, W), R.Tensor((n, 64), "float32"))

            gv0 = R.relu(lv0)
            R.output(gv0)
        return gv0
```

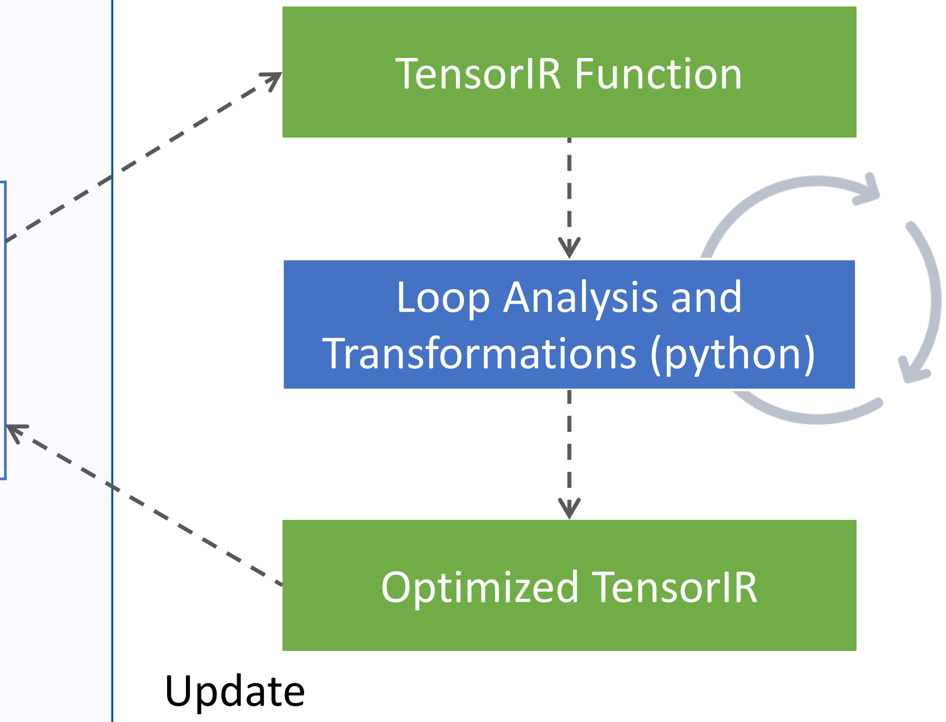
TensorIR functions
Loops, thread blocks

Call into TensorIR function via
destination passing

Analysis based Program Optimization

IRModule

```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class IRModule:
    @T.prim_func
    def mm(
        X: T.Buffer(("n", 128), "float32"),
        W: T.Buffer((128, 64), "float32"),
        Y: T.Buffer(("n", 64), "float32")
    ):
        n = T.int64()
        for i, j, k in T.grid(n, 64, 128):
            Y[i, j] += X[i, k] * W[k, j]


    @R.function
    def main(
        X: R.Tensor(("n", 128), "float32"),
        W: R.Tensor((128, 64), "float32")
    ):
        n = T.int64()
        with R.dataflow():

            lv0 = R.call_tir(mm, (X, W), R.Tensor((n, 64), "float32"))

            gv0 = R.relu(lv0)
            R.output(gv0)
        return gv0
```



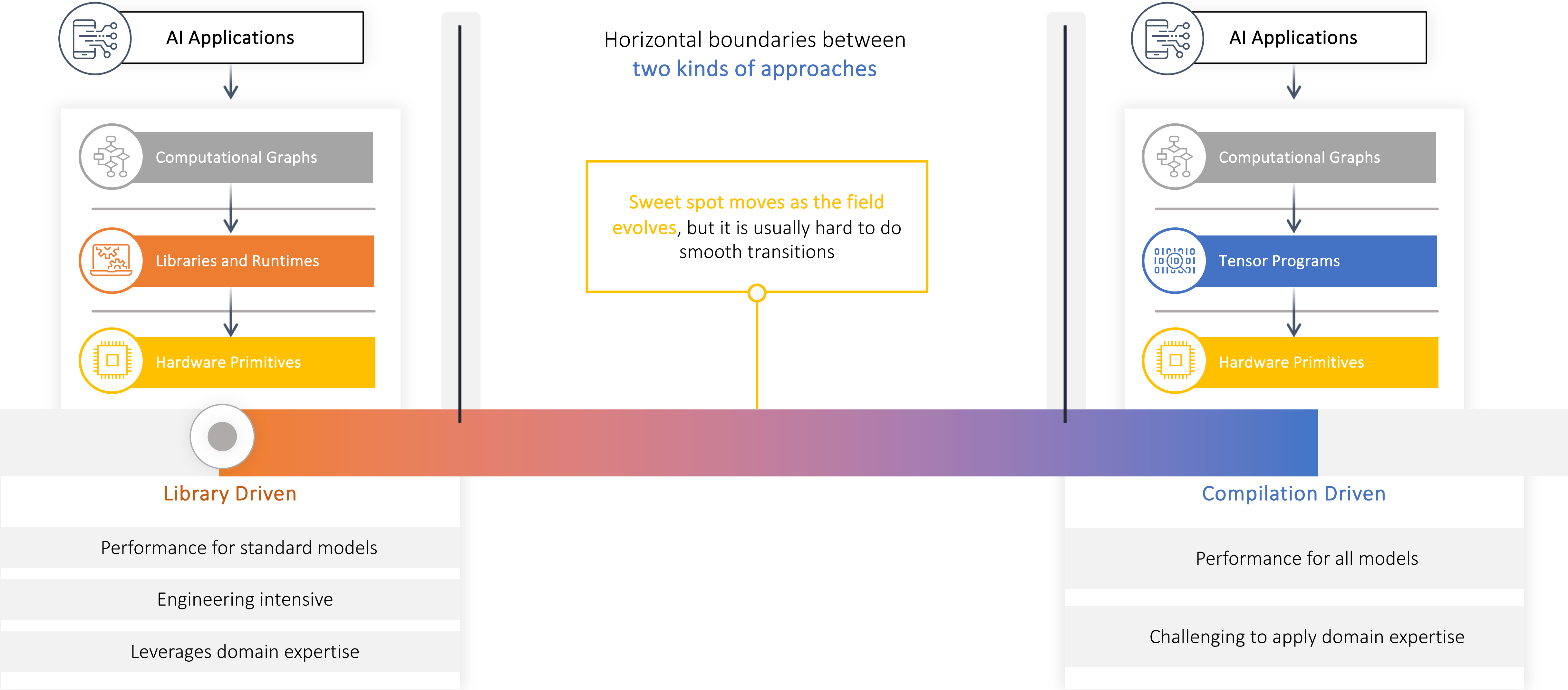
Abstraction Elements of TVM Unity

First-class symbolic shape support

Composable Tensor Program Optimization

Unifying Libraries and Compilation

Bringing Compilation and Libraries Together



Abstraction to Unify Libraires and Compilation

IRModule

```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class Module:
    @R.function
    def vae(
        data: R.Tensor(("n", 4, 64, 64), "float32"),
        params: R.Tuple(R.Tensor((4, 4, 1, 1), "float32"),
                        R.Tensor((1, 4, 1, 1), "float32"),
                        ...)
    ) -> R.Tensor(("n", 512, 512, 3),"float32"):
        n = T.int64()
        with R.dataflow():
            w0: R.Tensor((4, 4, 1, 1), "float32") = params[0]

            lv0: R.Tensor((n, 4, 64, 64), "float32") = R.call_dps_packed(
                "cutlass_conv2d", w0, R.Tensor((n, 4, 64, 64), "float32")
            )

            lv1: R.Tensor((n, 4, 64, 64), "float32") = R.add(lv0, b0)
            ...
```

Library Embedded via DLPack

```
void CutlassConv2D(
   DLTensor* input,
   DLTensor*output
) {
   ...
}

TVM_REGISTER_GLOBAL("cutlass_conv2d")
.set_body(CutlassConv2D);
```

Call into runtime library function registered via TVM FFI

Unify Libraires and Compilation

The fused conv_add operator is defined with Relax-BYOC offloading to TensorRT, a library with optimized kernels for Nvidia GPUs.

Relax-BYOC replaces all instances of conv_add with direct calls to TensorRT, while retaining the overall structure of the module.

```
@tvm.script.ir_module
class MyMod:
    @R.function
    def conv_add(x: R.Tensor(("n", 4, 64, 64)),
                 w: R.Tensor((4, 4, 1, 1)),
                 b0: R.Tensor((1, 4, 1, 1))):
        R.func_attrs({"codegen": "tensorrt"})
        gv0 = op.conv2d(x, w, padding=(1,1))
        gv1 = op.add(gv0, b0)
        return gv1

    @R.function
    def vae(data: R.Tensor(("n", 4, 64, 64), "float32"),
            params: R.Tuple(...)
    ) -> R.Tensor(("n", 512, 512, 3), "float32"):
        n = T.int64()
        with R.dataflow():
            lv1: R.Tensor((n, 4, 64, 64), "float32") =
                conv_add(data, params[0], params[1])
```

Offloading Pass
→

```
@tvm.script.ir_module
class MyMod:
    @R.function
    def vae(data: R.Tensor(("n", 4, 64, 64), "float32"),
            params: R.Tuple(...)
    ) -> R.Tensor(("n", 512, 512, 3), "float32"):
        n = T.int64()
        with R.dataflow():
            lv1: R.Tensor((n, 4, 64, 64), "float32") =
                call_dps_packed("conv_relu_tensorrt",
                                data, params[0],
                                params[1], R.Tensor(
                                (n, 4, 64, 64), "float32"))
```

Unify Libraires and Compilation

Bringing **library-based offloading** and **native compilation** together

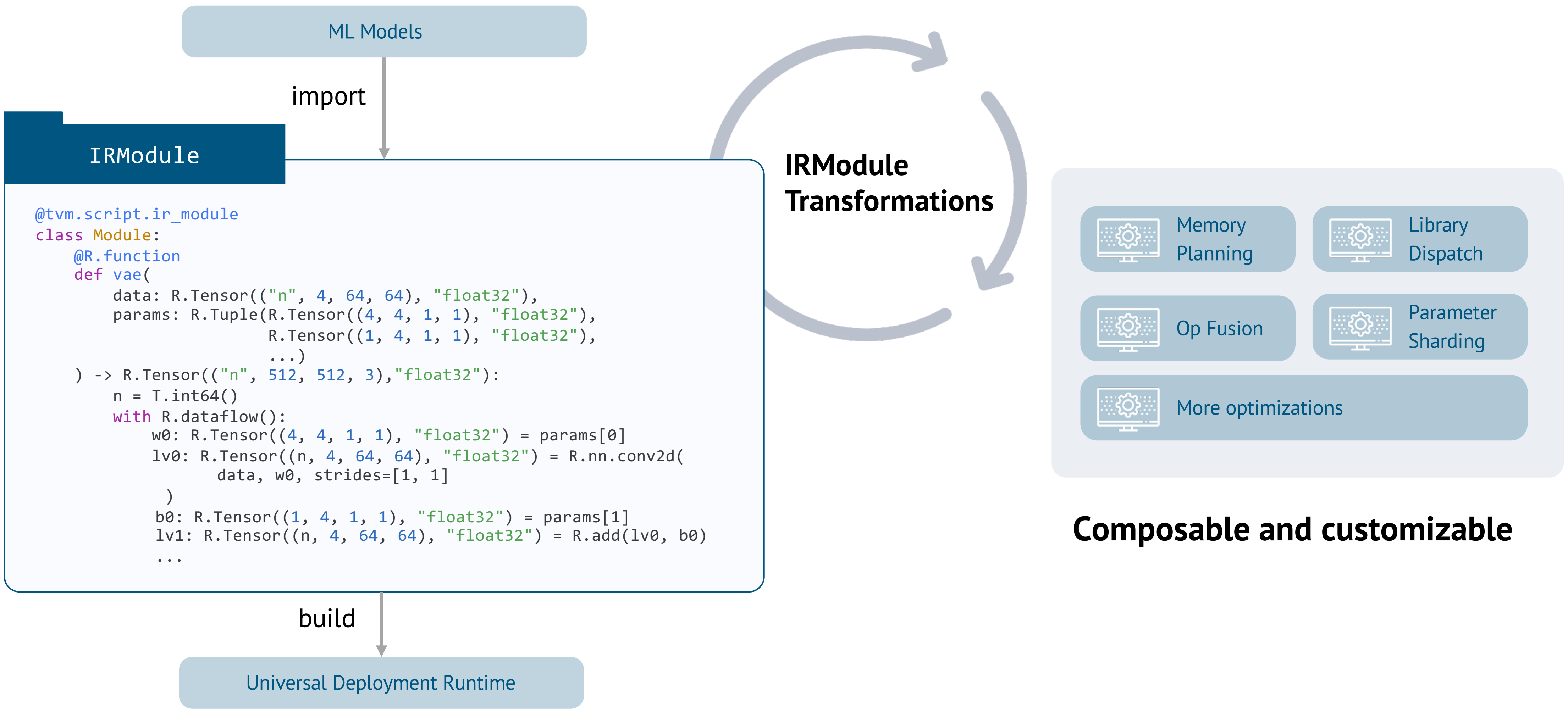
```
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class MyMod:
    @R.function
    def vae(data: R.Tensor(("n", 4, 64, 64), "float32"),
            params: R.Tuple(...)
    ) -> R.Tensor(("n", 512, 512, 3), "float32"):
        n = T.int64()
        with R.dataflow():
            lv1: R.Tensor((n, 4, 64, 64), "float32") =
                call_dps_packed("conv_relu_cutlass",,
                                 data, params[0], params[1],
                                 R.Tensor((n, 4, 64, 64), "float32"))
            w1: R.Tensor((512, 4, 3, 3), "float32") = params[2]
            lv2: R.Tensor((n, 512, 64, 64), "float32") = R.nn.conv2d(
                     lv1, w1, strides=[1, 1]
                )
```

Library Offloading

Native Compilation

ML Compilation in Action

Productive Framework for ML Compilation



Enabling Incremental Developments

New model or backend

```
mod = frontend.from_fx(model)
mod = relax.get_pipeline()(mod)
```

- Part of the model accelerated
- Find room for improvements

Composable customizations

Mix your own library and compilation

```
mod = DispatchToLibary("attention")(mod)
mod = DefaultTIRLegalization(mod)
```

Try out new fusion patterns

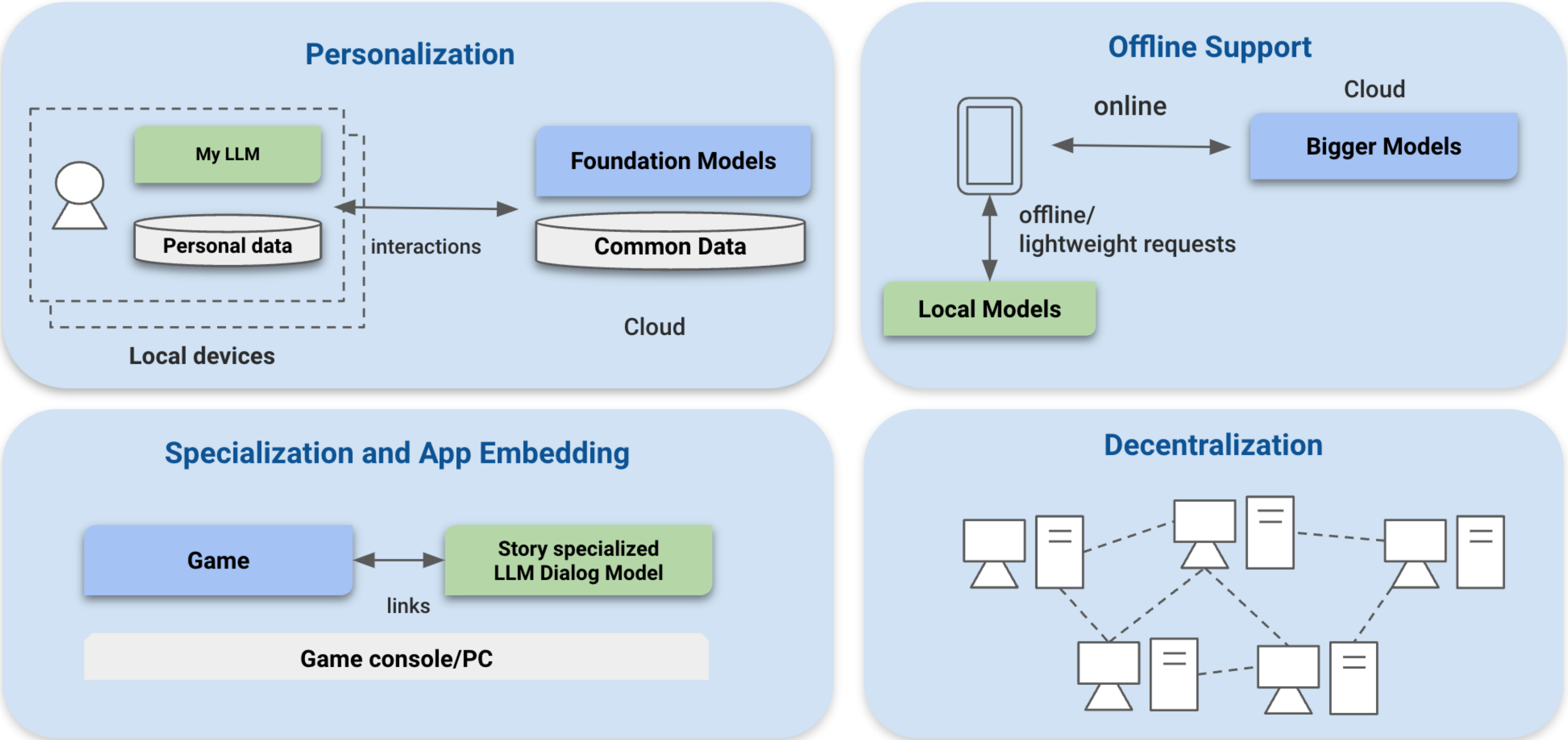
```
mod = CustomizeFusion()(mod)
mod = transform.Sequential([
    transform.FuseOps(),
    transform.FuseTIR()
])(mod)
```

Milestones and Feedbacks

- Feedback to out of box pipelines
- Full model accelerated and offloaded to target env
- Deploy ML compilation improvements to prod.

This is not a one shot game, but continuous process for every new model, backend features, new improvements in machine learning compilation.

Bringing foundational models to consumer devices



Challenges of Deploying Large Models to Consumer Hardware

Consumer Devices and Hardware Backends

GPU Runtimes

AMD ROCm

Vulkan

OpenCL

NVIDIA CUDA

WebGPU

Microsoft DirectX

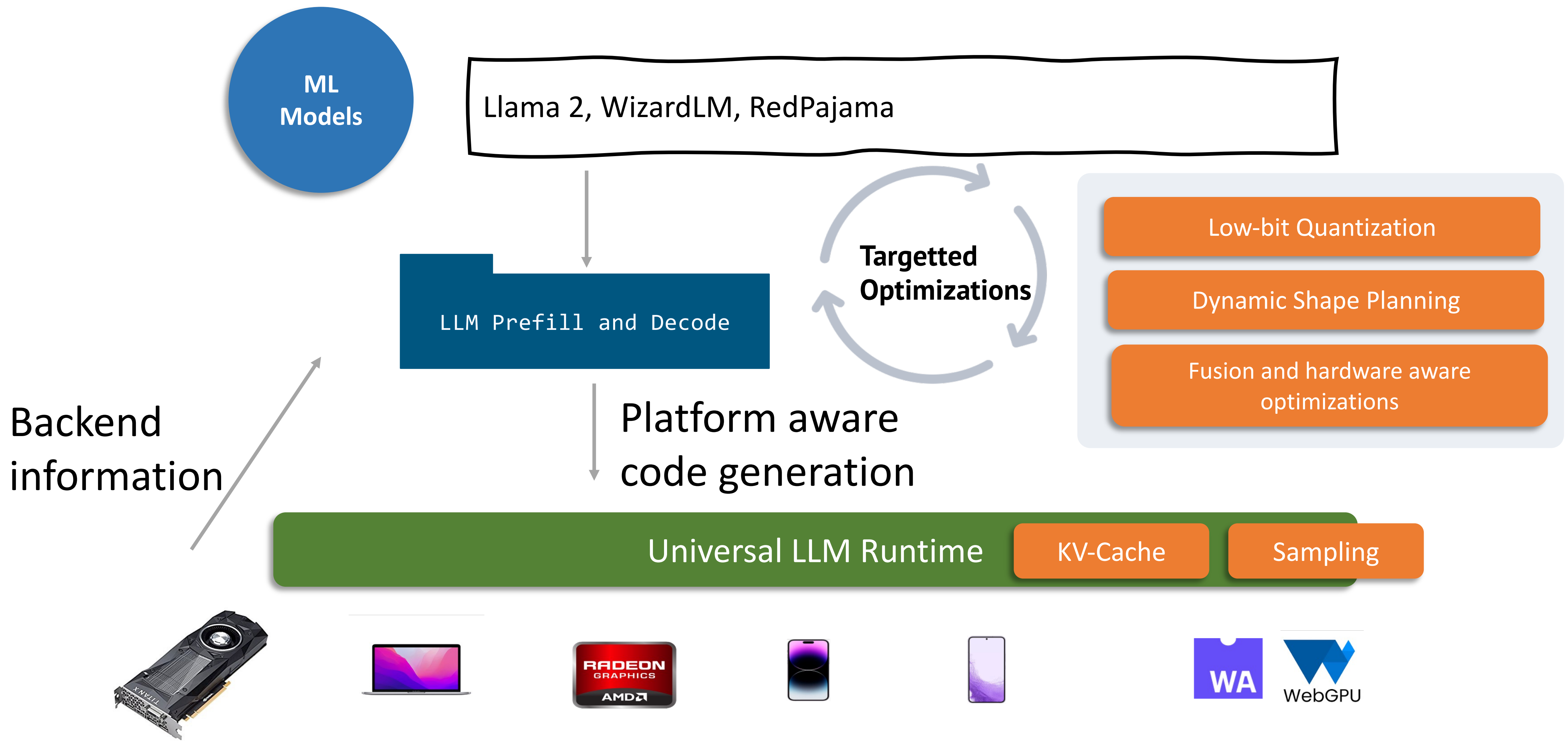
SYCL

...

Diversity of hardware and software stack

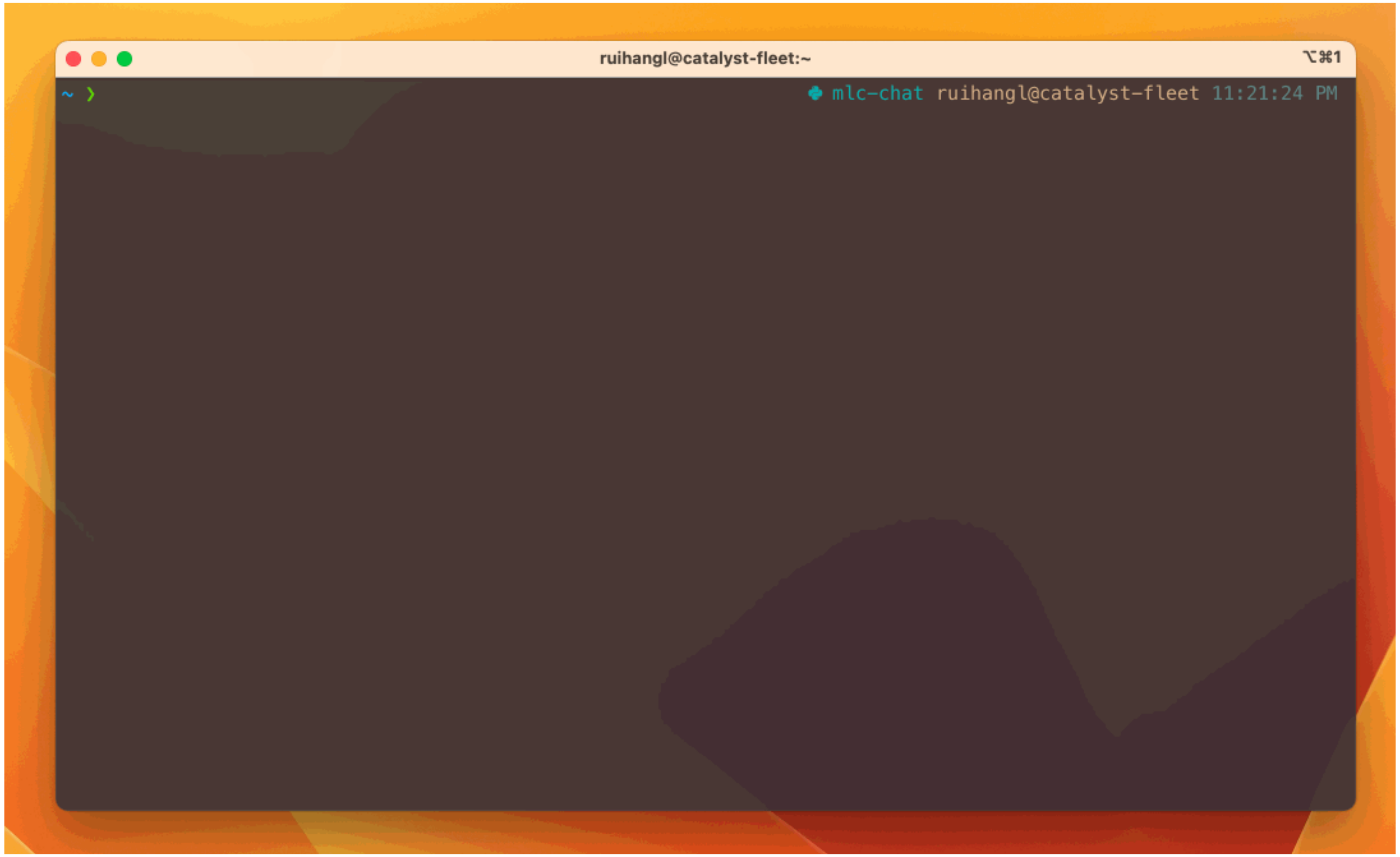
Continuous demand of machine learning system developments

ML Compilation can help



MLC LLM: Windows Linux Mac

Generalizes to Llama2 70B!



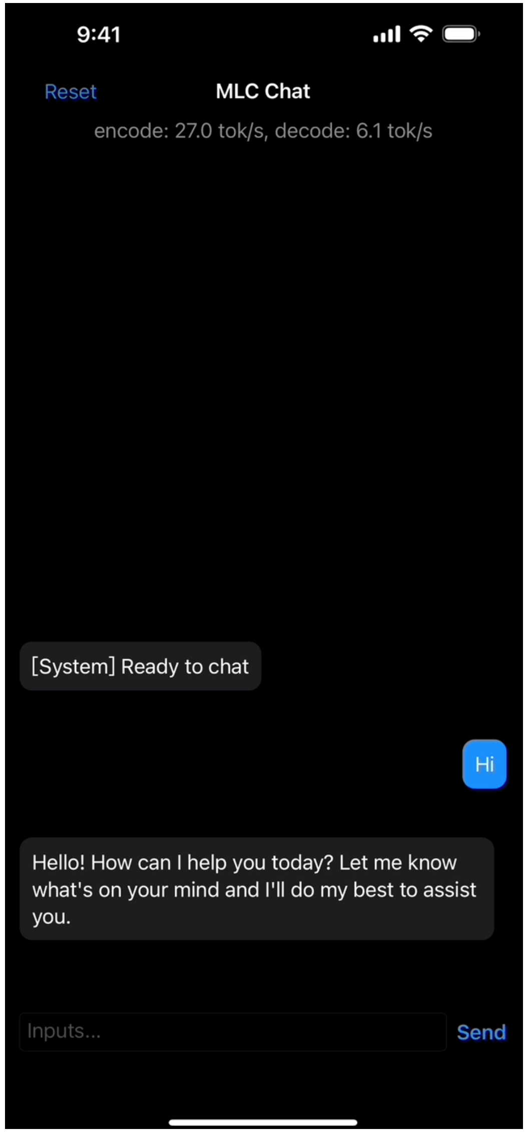
https://llm.mlc.ai/

MLC Chat: iPhone

Need iPhone with 6GB memory

Available on AppStore

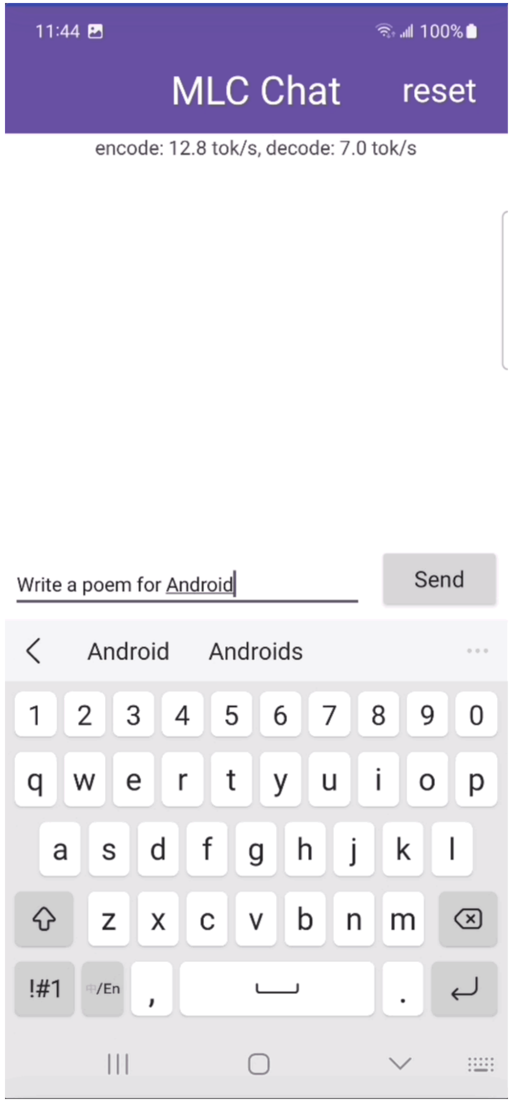
Search for MLC Chat



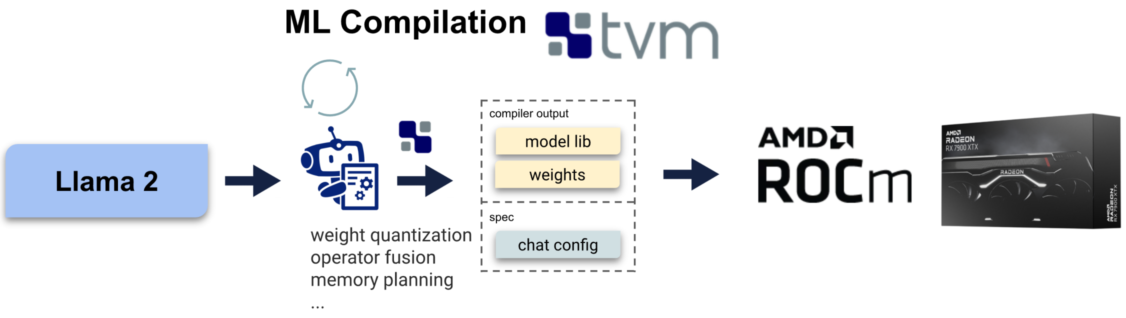
MLC LLM: Android

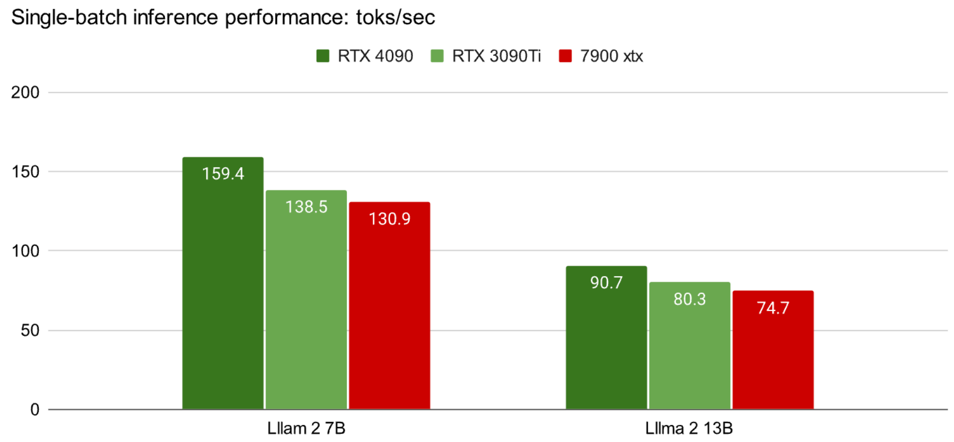
Snapdragon Gen2

Enables larger models than iPhone



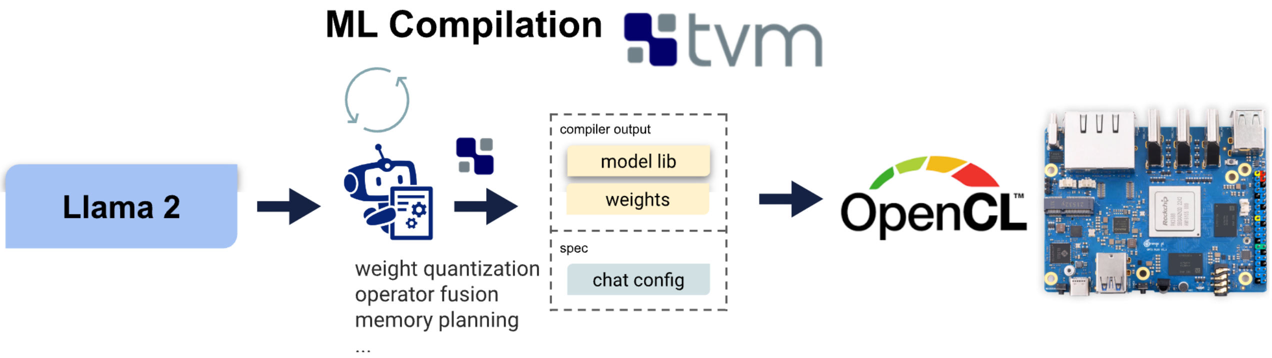
Making AMD GPUs competitive for LLM inference

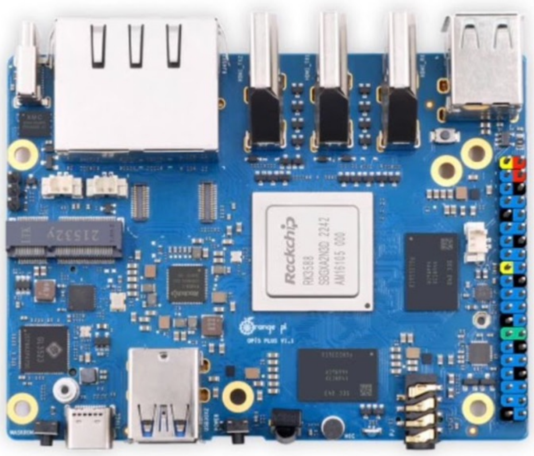


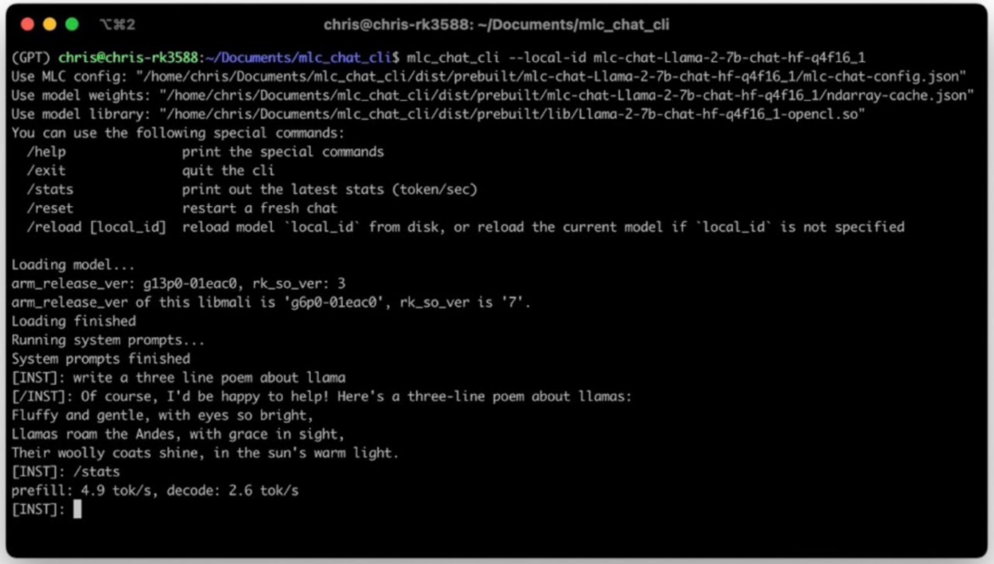


One day effort after connecting to rocm stack

Bringing LLMs to 100$ Orange Pi







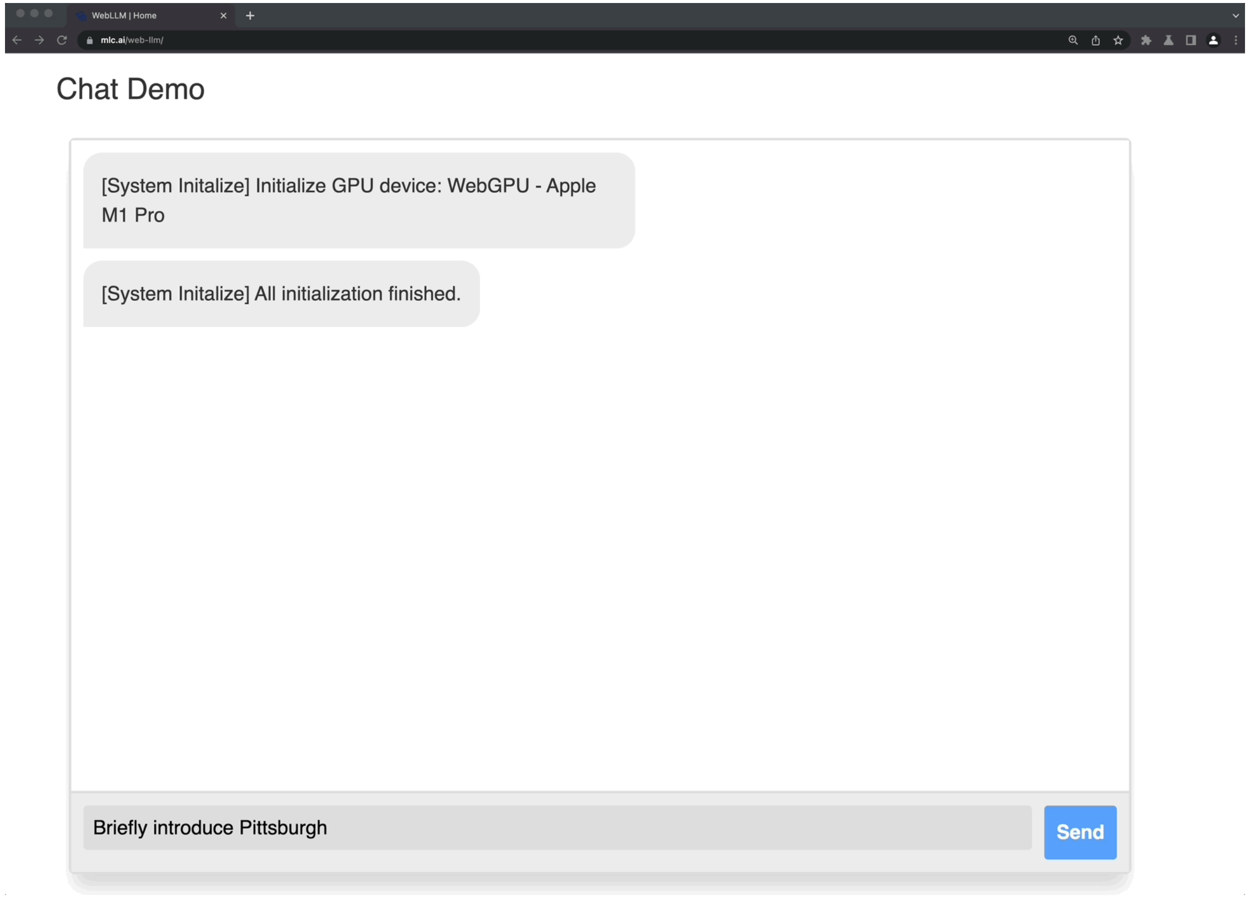
Web LLM

Runs directly in browser client

WebAssembly host support

Access native GPU from the browser with sandboxing

https://webllm.mlc.ai/



Information

MLSys Conference: mlsys.org

MLC LLM: llm.mlc.ai

MACHINE LEARNING COMPILATION mlc.ai